## Notice

Apple Computer, Inc. and Microsoft Corporation reserve the right to make improvements in the product described in this manual at any time and without notice.

## Disclaimer of All Warranties
## and Liability Relating to Software

**Apple III**

**Microsoft BASIC
Reference Manual**

# Contents

## 3 Microsoft BASIC Commands and Statements     34

**1**

# *Introduction*

# 1

# Introduction

Microsoft™ BASIC Release 5.2 is the most extensive implementation of BASIC available for microprocessors. It meets the requirements for the ANSI subset standard for BASIC and supports many features rarely found in other BASICs. In addition, Microsoft BASIC has sophisticated string handling and structured programming features that are especially suited for applications development. Microsoft BASIC gives users what they want from a BASIC — ease of use plus the features that make a microcomputer perform like a minicomputer or large mainframe.

In 1975, Microsoft wrote the first BASIC interpreter for microcomputers. Today, Microsoft BASIC, with over 750,000 installations in over 20 operating environments, is recognized as the defacto industry standard. It's the BASIC you will find on all of the most popular microcomputers. Many users, manufacturers, and software vendors have written application programs in Microsoft BASIC.

## Using This Manual

This manual has been specially prepared for use with Microsoft BASIC Release 5.2, which is included in your SoftCard™ III package. It serves as both a user's guide and technical reference, documenting both general information and detailed descriptions of the commands, statements, and functions.

This manual is not intended as a tutorial on BASIC. It is assumed that you have a working knowledge of the BASIC language. If you need more information on BASIC programming, refer to the "Resources for Learning BASIC" section in this chapter.

This manual contains the following information:

*Chapter 1   Introduction*

Provides a brief description of the contents of this manual, the notation used in describing BASIC language syntax, and a list of references for learning BASIC programming.

*Chapter 2   General Information about Microsoft BASIC*

Gives general information on loading BASIC, modes of operation, program format, special characters, data representation, and input editing.

*Chapter 3   Microsoft BASIC Commands and Statements*

Contains descriptions of all the commands and statements in Microsoft BASIC. The descriptions include syntax, what the command or statement is used for, and in most cases, examples.

*Chapter 4   Microsoft BASIC Functions*

Describes Microsoft BASIC functions. Descriptions include syntax, purpose, and examples.

*Appendix A   Converting Programs to Microsoft BASIC*

Shows how to convert a program written in another BASIC to Microsoft BASIC.

*Appendix B   Microsoft BASIC Disk I/O*

Explains disk I/O procedures for the user who is unfamiliar with disk I/O conventions and routines.

*Appendix C   BASIC Assembly Language Subroutines*

Discusses how to interface to assembly language subroutines with the USR function and the CALL statement.

*Appendix D   Summary of Error Codes and Error Messages*

Presents a detailed description of all the error messages in Microsoft BASIC and their possible causes.

*Appendix E   Mathematical Functions*

Provides a set of formulas for math functions that are not built into Microsoft BASIC.

*Appendix F   ASCII Character Codes*

Shows the different values of the ASCII character set.

*Appendix G   Microsoft BASIC Reserved Words*

Lists all the reserved words in Microsoft BASIC.

## Syntax Notation

The following notation is used throughout this manual in descriptions of command and statement syntax:

[ ]   Square brackets indicate that the enclosed entry is optional.

< >   Angle brackets indicate user entered data. When the angle brackets enclose lowercase text, the user must type in an entry defined by the text; for example, <filespec>. When the angle brackets enclose uppercase text, the user must press the key named by the text; for example, <RETURN>.

Since the Apple™ III computer displays lowercase characters, you may enter commands, statements, and functions in lowercase form. This will not affect operation.

{ }   Braces indicate that the user has a choice between two or more entries. At least one of the entries enclosed in braces must be chosen unless the entries are also enclosed in square brackets.

. . .   Ellipses indicate that an entry may be repeated as many times as needed or desired.

CAPS   Capital letters indicate portions of statements or commands that must be entered, exactly as shown.

All other punctuation, such as commas, colons, slash marks, and equal signs must be entered exactly as shown.

## Resources for Learning BASIC

This manual provides complete instructions for using Microsoft BASIC. However, no teaching material for BASIC programming has been provided. If you are new to BASIC or need help in learning programming, we suggest you read one of the following:

Albrecht, Robert L.; Finkel, LeRoy; Brown, Jerry. *BASIC.* John Wiley Sons, 1973.

Dwyer, Thomas A. and Critchfield, Margot. *BASIC and the Personal Computer.* Addison-Wesley Publishing Co., 1978.

Simon, David E. *BASIC From the Ground Up.* Hayden, 1978.

**2**

# General Information about Microsoft BASIC

# 2

# *General Information about Microsoft BASIC*

## *Initialization*

Your SoftCard III package includes the CP/M™ version of Microsoft BASIC (MBASIC) on a 5.25 inch single density diskette. The name of the file is MBASIC.COM.

To load and run MBASIC with the default memory configuration, bring up CP/M and wait for the A> prompt. Once the prompt appears, type the following:

MBASIC <RETURN>

The system will reply:

BASIC-80 Rev 5.2
[CP/M Version]
Copyright 1977, 78, 79, 80 ⓒ by Microsoft
Created dd-mon-yy
xxxxx Bytes free
Ok

The default memory configuration sets the number of files that may be open at one time during execution of a BASIC program to three. It also sets the maximum record size at 128 bytes and allows the use of all RAM memory up to the start of FDOS (an arbitrary area in memory set by CP/M). See Chapter 2 of the *CP/M Reference Manual* for more information.

If you wish to change the memory configuration, the following command line format can be used in place of the simple MBASIC command for initialization.

MBASIC [<filespec>] [/F:<number of files>]
[/M:<highest memory location>] [/S:<maximum record size>]

The <filespec> option allows you to RUN a program after initialization is complete. <filespec> consists of a filename and optional filename extension. A default extension of .BAS is used if none is supplied and the filename is less than nine characters long. This allows BASIC programs to be executed in batch mode using the SUBMIT facility of CP/M. Such programs should include a SYSTEM statement (see Chapter 3) to return to CP/M command level when they have finished, allowing the next program in the batch stream to execute.

The /F:<number of files> option sets the number of disk files that may be open at any one time during the execution of a BASIC program. Each file data block allocated in this fashion requires 166 bytes plus 128 bytes (or the number specified by the /S: option) of memory. If the /F option is omitted, the number of files defaults to 3. The <number of files> may be entered in a decimal form (default condition), octal form (preceded by an &O) or hexadecimal form (preceded by an &H).

The /M:<highest memory location> option sets the highest memory location that will be used by MBASIC. In some cases, it is desirable to set the amount of memory well below the FDOS area to reserve space for assembly language subroutines. In all cases, <highest memory location> should be below the start of FDOS (whose address is contained in locations 6 and 7). If the /M option is omitted, all memory up to the start of FDOS is used.

The /S:<maximum record size> option sets the maximum record size for use with random files. Any whole number may be specified, including numbers larger than 128 (the default record size).

Here are a few examples of the different initialization options:

A>MBASIC PAYROLL.BAS          Use all memory and three files,
                              load and execute PAYROLL.BAS.

A>MBASIC INVENT/F:6           Use all memory and six files,
                              load and execute INVENT.BAS.

A>MBASIC /M:32768             Use first 32K of memory and three
                              files.

A>MBASIC DATACK/F:2/M:&H9000
                              Use first 36K of memory, two
                              files, and execute DATACK.BAS.

To return to CP/M, use the SYSTEM command. SYSTEM closes all files and then performs a CP/M warm start (reboots CP/M).

# Modes of Operation

When Microsoft BASIC is initialized, the prompt "Ok" is displayed. "Ok" means BASIC is at command level; that is, it is ready to accept commands. At this point, Microsoft BASIC may be used in either of two modes: the direct mode or the indirect mode.

In the direct mode, BASIC statements and commands are not preceded by line numbers. They are executed as they are entered. Results of arithmetic and logical operations may be displayed immediately and stored for later use, but the instructions themselves are lost after execution. This mode is useful for debugging and for using BASIC as a "calculator" for quick computations that do not require a complete program.

The indirect mode is used for entering programs. Program lines are preceded by line numbers and are stored in memory. The program stored in memory is executed by entering the RUN command.

# File Naming Conventions

Disk files are described by their file specification or *filespec*, for short. Filespecs are a string expression of the form:

[disk identifier:]<filename>[.filename extension]

The disk identifier instructs BASIC where to look for the file, and the filename tells BASIC which file to look for. The filename extension is a label that tells BASIC what type the file is. Filename is the only required parameter. The disk identifier and filename extension are optional. Each part of the filespec is discussed in the following paragraphs.

## Filename

The filename may be from one to eight characters in length and may consist of either uppercase or lowercase alphanumeric characters, or a combination of both. CP/M will not recognize filenames longer than eight characters. Examples of valid filenames:

PAYROLL      ACNT4      A2400      Barb

Certain special characters and all control characters cannot be used as filenames. These characters are:

= ? * < > . , ; : [ ]

CP/M uses these characters in other ways. Therefore, they cannot be used as filenames.

## Filename Extensions

A filename extension identifies the type of a file. For example, .ASM identifies an assembly language source file, whereas .BAS identifies a BASIC program source file. Filename extensions in CP/M are from one to three characters in length and are preceded by a period. The filename can be made up of letters or alphabetic characters, or a combination of both. Most often, you will use one

of the de facto standard extensions as shown in the list below.

| | |
|---|---|
| .ASM | Assembly language source file |
| .BAK | Backup file |
| .BAS | BASIC source file |
| .COM | Command file |
| .DAT | Data file |
| .DOC | Text (document file) |
| .HEX | Intel HEX format object code file |
| .LIB | Library file |
| .MAC | Macro file (usually a subroutine used in assembly language programs) |
| .OBJ | Machine code (object file) |
| .PRN | Assembly language list file (PRINT file) |
| .REL | Relocatable machine code program file |
| .TXT | Text file |

Although other extensions may be used, this list represents the majority of extensions you will use with CP/M.

The most common extension you will use is .BAS. .BAS is used as a default extension when LOAD, SAVE, MERGE, or RUN commands are executed (if no other extension is given and the filename is less than nine characters long). Some examples of filename extensions:

APPLE3.TXT    ACCReciv.BAS    PROGRAM.4    POLS.C12

## Disk Drive Identifiers

Disk drives are identified in CP/M by letters. The first drive is the *primary drive* and is always identified with the letter A. Other drives follow in alphabetical order.

Disk drive identifiers precede the filename and consist of the identifying letter (A-D) and a colon (:). The colon separates the disk drive identifier from the filename.

If no identifier is specified, CP/M assumes the default drive (unless otherwise specified, the default drive is always drive A). For example:

A:PROGRAM.BAS.

## Line Format

Program lines in a BASIC program have the following format:

nnnnn BASIC statement [:BASIC statement...] [comment]

Program lines are ended by pressing the <RETURN> key. More than one BASIC statement may be placed on a line, but each statement on a line must be separated from the last by a colon.

A Microsoft BASIC program line always begins with a line number and ends with a carriage return. It may contain a maximum of 255 characters.

It is possible to extend a logical line over more than one physical line by using Control-J. Control-J lets you continue typing a logical line on the next physical line without entering <RETURN>.

## Line Numbers

Every BASIC program line begins with a line number. Line numbers indicate the order in which the program lines are stored in memory. They are also used as references in branching and editing. Line numbers must be in the range 0 to 65529. A period (.) may be used in EDIT, LIST, AUTO, and DELETE commands to refer to the current line.

## Character Set

The Microsoft BASIC character set is comprised of alphabetic characters, numeric characters, and special characters. These are the characters that BASIC recognizes. There are many others which can be displayed or printed but have no particular meaning to BASIC. See Appendix F, "ASCII Character Codes," for a complete list of all these characters.

The alphabetic characters in Microsoft BASIC are the uppercase and lowercase letters of the alphabet.

The numeric characters in Microsoft BASIC are the digits 0 through 9.

The following special characters and terminal keys are recognized by Microsoft BASIC:

| Character | Name |
|---|---|
| | Blank |
| = | Equal sign or assignment symbol |
| + | Plus sign |
| - | Minus sign |
| * | Asterisk or multiplication symbol |
| / | Slash or division symbol |
| ^ | Up arrow or exponentiation symbol |
| ( | Left parenthesis |
| ) | Right parenthesis |
| % | Percent |
| # | Number (or pound) sign |
| $ | Dollar sign |
| ! | Exclamation point |
| [ | Left bracket |
| ] | Right bracket |
| , | Comma |
| . | Period or decimal point |
| ' | Single quotation mark (apostrophe) |
| ; | Semicolon |
| : | Colon |
| & | Ampersand |
| ? | Question mark |
| < | Less than |
| > | Greater than |
| \ | Backslash or integer division symbol |
| @ | At-sign |
| — | Underscore |
| <ESCAPE> | Escapes Edit Mode subcommands. (See Chapter 3) |
| <TAB> | Moves print position to next tab stop. Tab stops are set every eight columns. |
| <RETURN> | Terminates input of a line. |

## Control Characters

The following control characters are used in Microsoft BASIC:

| Control Character | Description |
|---|---|
| Control-A | Enters Edit Mode on the line being typed. |
| Control-B | Backslash. |
| Control-C | Interrupts program execution and returns to BASIC command level. |
| Control-G | Rings the bell (a beep from the speaker) at the console. |
| Control-H | Backspace. Deletes the last character typed. This is the same as the ← key. |
| Control-I | Tab. Tab stops are set at every eight columns. Same as →. |
| Control-J | Line Feed. Moves to the next physical line. |
| Control-K | Right square bracket. |
| Control-O | Halts program output while execution continues. A second Control-O restarts output. |
| Control-Q | Resumes program execution after a Control-S. |
| Control-R | Repeats the line that is currently being typed. |
| Control-S | Suspends program execution. |

**Control Characters** *(continued)*

| Control Character | Description |
| --- | --- |
| Control-U | Deletes the line that is currently being typed. |
| Control-X | Same as Control-U. |
| Control-Y | Permits recovery after pressing RESET. |

# *Reserved Words*

Reserved words are words that have special meaning in Microsoft BASIC. They include all BASIC commands, statements, function names, and operator names.

You should always separate reserved words from data or other elements of a BASIC statement with spaces or other special characters as allowed by the syntax. In addition, reserved words may not be used for variable names.

A complete list of reserved words is given in Appendix G.

# *Constants*

Constants are the actual values BASIC uses during execution. There are two types of constants: string and numeric.

A string constant is a sequence of up to 255 alphanumeric characters enclosed in double quotation marks. Examples of string constants:

> "HELLO"
> "$25,000.00"
> "Number of Employees"

Numeric constants are positive or negative numbers. Numeric constants in BASIC cannot contain commas. There are five types of numeric constants:

| | |
| --- | --- |
| Integer constants | Whole numbers between -32768 and +32767. Integer constants do not contain decimal points. |
| Fixed-point constants | Positive or negative real numbers, i.e., numbers that contain decimal points. |
| Floating-point constants | Positive or negative numbers represented in exponential form (similar to scientific notation). A floating-point constant consists of an optionally signed integer or fixed-point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). The allowable range for floating point constants is $10^{-38}$ to $10^{+38}$. Examples: |

> 235.988E-7 = .0000235988
> 2359E6 = 2359000000

(Double precision floating-point constants are denoted by the letter D instead of E.)

| | |
| --- | --- |
| Hex constants | Hexadecimal numbers with the prefix &H. Examples: |

> &H76
> &H32F

| | |
| --- | --- |
| Octal constants | Octal numbers with the prefix &O or &. Examples: |

> &O347
> &1234

## *Single and Double Precision Numeric Constants*

Numeric constants may be either single precision or double precision numbers. Single precision numeric constants are stored

with seven digits of precision and printed with up to six digits. Double precision numbers are stored with sixteen digits of precision and printed with up to sixteen digits.

A single precision constant is any numeric constant that has one of the following properties:

1. Seven or fewer digits

2. Exponential form using E

3. A trailing exclamation point (!)

A double precision constant is any numeric constant that has one of the following properties:

1. Eight or more digits

2. Exponential form using D

3. A trailing number sign (#)

| Single Precision Constants | Double Precision Constants |
|---|---|
| 46.8 | 345692811 |
| −1.09E-06 | −1.09432D-06 |
| 3489.0 | 3489.0# |
| 22.5! | 7654321.1234 |

# Variables

Variables are names which represent values that are used in a program. As with constants, there are two types of variables: numeric and string. A numeric variable may be assigned a value that is a number. A string variable may only be assigned a character string value. The value of the variable may be assigned by the user or it may be assigned as the result of calculations in the program. In either case, the variable must always match the type of data that is assigned to it.

Before a variable is assigned a value, its value is assumed to be zero (numeric variables) or null (string variables).

## Variable Names

Microsoft BASIC variable names may contain up to 255 characters. However, only the first 40 characters are significant. The characters allowed in a variable name are letters, numbers, and the decimal point. The first character in a variable name must be a letter. Special type declaration characters are also allowed (see the next section).

A variable name may not be a reserved word, but embedded reserved words are allowed. If a variable begins with FN, it is assumed to be a call to a user-defined function. (See "DEF FN," in Chapter 3 for more information on user-defined functions). A variable name may not be a reserved word with one of the type declaration characters ($, %, !, #) at the end.

For example,

    10 LOG = 8

is illegal, because LOG is a reserved word. Reserved words include all Microsoft BASIC commands, statements, function names, and operator names.

## Declaring Variable Types

Variable names may declare either a numeric value or a string. String variable names are written with a dollar sign ($) as the last character. For example:

    A$ = "SALES REPORT."

The dollar sign is a variable type declaration character; that is, it "declares" that the variable will represent a string.

Numeric variable names may declare integer, single, or double precision values. Computations with integer and single precision variables are less accurate than those with double precision

variables. However, you may want to declare a variable to a lower precision type because:

1.  Variables of higher precision take up more memory space. This is important if memory space is limited.

2.  Arithmetic computation times are longer for higher precision numbers. A program with repeated calculations runs faster with integer variables.

The type declaration characters for numeric variables and the memory requirements (in bytes) to store each variable type are as follows.

| Declaration Character | Variable Type | Bytes Required |
|---|---|---|
| % | Integer | 2 |
| ! | Single precision | 4 |
| # | Double precision | 8 |
| $ | String | 3 bytes overhead plus the present contents of the string. |

The default type for a numeric variable is single precision.

Examples of Microsoft BASIC variable names:

| | |
|---|---|
| PI# | Declares a double precision value. |
| MINIMUM! | Declares a single precision value. |
| LIMIT% | Declares an integer value. |
| N$ | Declares a string value. |
| ABC | Represents a single precision value. |

There is a second method by which variable types may be declared. The BASIC statements DEFINT, DEFSTR, DEFSNG, and DEFDBL may be included in a program to declare the types for certain variable names. These statements are described in detail in the "DEFINT/SNG/DBL/STR" section in Chapter 3.

## Array Variables

An array is a group or table of values referenced by the same variable name. The individual values in an array are called elements. Array elements are variables. They can be used in any BASIC statement or function which uses variables. Declaring the name and type of an array and setting the number of elements in the array is known as *defining* or *dimensioning* the array.

Each array element is referenced by an array variable that is subscripted with an integer or an integer expression. An array variable name has as many subscripts as there are dimensions in the array. For example, V(10) would reference a value in a one-dimension array, T(1,4) would reference a value in a two-dimension array, and so on. Note that the array variable T and the variable T are not the same variable. The maximum number of dimensions for an array is 255. The maximum number of elements per dimension is 32767.

Array elements, like numeric variables, require a certain amount of memory space, depending on the variable type. The memory requirements to store arrays (in bytes) are as follows.

| Element Type | Bytes |
|---|---|
| Integer | Two per element |
| Single Precision | Four per element |
| Double Precision | Eight per element |

## ERR and ERL Variables

ERR and ERL are special read-only variables that return the error code and line number associated with an error. Read-only variables cannot be assigned values.

When an error handling subroutine is entered, the variable ERR contains the error code for the error, and the variable ERL contains the number of the line in which the error was detected. The ERR and ERL variables are usually used in IF...THEN statements (see Chapter 3) to direct program flow in the error trap routine.

If the statement that caused the error was a direct mode statement, ERL will contain 65535. To test if an error occurred in a direct statement, use

        IF 65535 = ERL THEN ...

Otherwise, use

        IF ERR = error code THEN ...

        IF ERL = line number THEN ...

If the line number is not on the right side of the relational operator, it cannot be renumbered by RENUM. Because ERL and ERR are reserved variables, neither may appear to the left of the equal sign in a LET (assignment) statement. Microsoft BASIC error codes are listed in Appendix D.

# *Type Conversion*

When necessary, BASIC will convert a numeric constant from one type to another. The following rules and examples should be kept in mind.

1.  If a numeric constant of one type is set equal to a numeric variable of a different type, the number will be stored as the type declared in the variable name. (If a string variable is set equal to a numeric value or vice versa, a "Type mismatch" error occurs.)

            10 A% = 23.42
            20 PRINT A%
            RUN
            23

2.  During expression evaluation, all of the operands in an arithmetic or relational operation are converted to the same degree of precision, i.e., that of the most precise operand.

Also, the result of an arithmetic operation is returned to this degree of precision.

| | |
|---|---|
| 10 D# = 6#/7# | The arithmetic was performed |
| 20 PRINT D# | in double precision and the |
| RUN | result was returned in D# as a |
| .8571428571428571 | double precision value. |

### *Note*

Both operands must be double precision variables. If one of the variables is a single precision variable, then the last eight digits in the result are meaningless.

| | |
|---|---|
| 10 D = 6#/7 | The arithmetic was performed |
| 20 PRINT D | in double precision (and the |
| RUN | result was returned to D a |
| .857143 | single precision variable), |
| | rounded, and printed as a |
| | single precision value. |

3.  Logical operators (see the next section) convert their operands to integers and return an integer result. Operands must be in the range -32768 to 32767 or an "Overflow" error occurs.

4.  When a floating-point value is converted to an integer, the fractional portion is rounded.

            10 C% = 55.88
            20 PRINT C%
            RUN
            56

5.  If a double precision variable is assigned a single precision value, only the first seven digits (rounded) of the converted number will be valid, since only seven digits of accuracy were supplied with the single precision value. The absolute value of the difference between the printed double precision number and the original single precision value will be less

than 6.3E-8 times the original single precision value.

```
10 A = 2.04
20 B# = A
30 PRINT A;B#
RUN
 2.04  2.039999961853027
```

# Expressions and Operators

An expression may be a string or numeric constant, a variable, or a single value obtained by combining a constant and a variable with an operator.

An operator performs mathematical or logical operations on values. The operators provided by BASIC may be divided into four categories:

1.  Arithmetic

2.  Relational

3.  Logical

4.  Functional

## Arithmetic Operators

The arithmetic operators, in order of operational precedence, are listed in the following table.

| Operator | Operation | Sample Expression |
|---|---|---|
| ^ | Exponentiation | X^Y |
| — | Negation | —X |
| *,/ | Multiplication, Floating-point division | X*Y<br>X/Y |
| +,— | Addition, Subtraction | X+Y |

To change the order in which the operations are performed, use parentheses. Operations within parentheses are performed first. Inside parentheses, the usual order of operations is maintained.

Here are some sample algebraic expressions and their BASIC counterparts.

| Algebraic Expression | BASIC Expression |
|---|---|
| $X+2Y$ | X+Y*2 |
| $X - \dfrac{Y}{Z}$ | X – Y/Z |
| $\dfrac{XY}{Z}$ | X*Y/Z |
| $\dfrac{X+Y}{Z}$ | (X+Y)/Z |
| $(X^2)Y$ | (X^2)*Y |
| $X^{\frac{Z}{Y}}$ | X^(Y^Z) |
| $X(-Y)$ | X*(−Y). Two consecutive operators must be separated by parentheses. |

## Integer Division and Modulo Arithmetic

Two additional operations are available in Microsoft BASIC: integer division and modulo arithmetic.

*Integer division*

Integer division is denoted by the backslash ( \ ). The operands are rounded to integers (must be in the range -32768 to 32767) before the division is performed, and the quotient is truncated to an integer. For example:

```
Ok
10 X = 10\4
20 Y = 25.68\6.99
30 PRINT X;Y
RUN
 2     3
Ok
```

Integer division immediately follows multiplication and floating-point division in the established order of operational precedence.

*Modulo Arithmetic*

Modulo arithmetic is denoted by the operator MOD. Modulo arithmetic provides the integer value that is the remainder of an integer division. For example:

10.4 MOD 4 = 2 (10\4 = 2 with a remainder 2)
25.68 MOD 6.99 = 5 (26\7 = 3 with a remainder 5)

Modulo arithmetic immediately follows integer division in the established order of operational precedence

# Overflow and Division by Zero

If, during the evaluation of an expression, a division by zero is encountered, the "Division by zero" error message is displayed, machine infinity with the sign of the numerator is supplied as the result of the division, and execution continues. If the evaluation of an exponentiation results in zero being raised to a negative power, the "Division by zero" error message is displayed, positive machine infinity (the highest number the computer can produce) is supplied as the result of the exponentiation, and execution continues.

If overflow occurs, the "Overflow" error message is displayed, machine infinity with the algebraically correct sign is supplied as the result, and execution continues.

# Relational Operators

Relational operators are used to compare two values. The result of the comparison is either "true" (-1) or "false" (0). This result may be then used to make a decision regarding program flow (see "IF . . . THEN," Chapter 3).

| Operator | Relation Tested | Expression |
|---|---|---|
| = | Equality | $X=Y$ |
| <> | Inequality | $X<>Y$ |
| < | Less than | $X<Y$ |
| > | Greater than | $X>Y$ |
| <= | Less than or equal to | $X<=Y$ |
| >= | Greater than or equal to | $X>=Y$ |

(The equal sign is also used to assign a value to a variable. See "LET," Chapter 3.)

When arithmetic and relational operators are combined in one expression, the arithmetic operation is always performed first. For example, the expression

$X+Y < (T-1)/Z$

is true if the value of X plus Y is less than the value of T-1 divided by Z.

IF SIN(X) < 0 GOTO 1000

IF I MOD J <> 0 THEN K=K+1

# Logical Operators

Logical operators perform tests on multiple relations, bit manipulations, or Boolean operations. Just as the relational operators can be used to make decisions regarding program flow, logical operators can connect two or more relations and return a true or false value to be used in a decision (see "IF...THEN," Chapter 3). For example:

IF D<200 AND F<4 THEN 80
IF I>10 OR K<0 THEN 50
IF NOT P THEN 100

A logical operator returns a result from a combination of true-false operands. The result (in bits) is either "true" (not zero) or "false" (zero). The true-false combinations and the results of a logical operation are known as truth tables.

There are six logical operators in Microsoft BASIC; they are: NOT (logical complement), AND (conjunction), OR (disjunction), XOR (exclusive or), IMP (implication), and EQV (equivalence). Each operator returns results as indicated in the following truth tables. A "T" indicates a true, or non-zero, value. "F" indicates a false, or zero, value. Operators are listed in order of precedence.

In an expression, logical operations are performed after arithmetic and relational operations.

## Table 1. Microsoft BASIC Logical Truth Tables

NOT

| X | NOT X |
|---|-------|
| 1 | 0 |
| 0 | 1 |

AND

| X | Y | X AND Y |
|---|---|---------|
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

OR

| X | Y | X OR Y |
|---|---|--------|
| 1 | 1 | 1 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

XOR

| X | Y | X XOR Y |
|---|---|---------|
| 1 | 1 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

***Table 1.*** *(Continued)*

IMP

| X | Y | X IMP Y |
|---|---|---------|
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 1 |
| 0 | 0 | 1 |

EQV

| X | Y | X EQV Y |
|---|---|---------|
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 1 |

## How Logical Operators Work

Logical operators convert their operands to sixteen bit, signed, two's complement integers in the range -32768 to +32767. (If the operands are not in this range, an error results.) If both operands are supplied as 0 or -1, logical operators return 0 or -1. The given operation is performed on these integers in bits, i.e., each bit of the result is determined by the corresponding bits in the two operands.

Thus, it is possible to use logical operators to test bytes for a particular bit pattern. For instance, the AND operator may be used to "mask" all but one of the bits of a status byte at a machine I/O port. The OR operator may be used to "merge" two bytes to create a particular binary value. The following examples will help demonstrate how the logical operators work.

| | |
|---|---|
| 63 AND 16=16 | 63 = binary 111111 and 16 = binary 10000, so 63 AND 16 = 16. |
| 15 AND 14=14 | 15 = binary 1111 and 14 = binary 1110, so 15 AND 14 = 14 (binary 1110). |
| -1 AND 8=8 | -1 = binary 1111111111111111 and 8 = binary 1000, so -1 AND 8 = 8. |
| 4 OR 2=6 | 4 = binary 100 and 2 = binary 10, so 4 OR 2 = 6 (binary 110). |

10 OR 10=10    10 = binary 1010, so 1010 OR 1010 = 1010 (10).

-1 OR -2 = -1    -1 = binary 1111111111111111 and 2 = binary 1111111111111110, so -1 OR -2 = -1. The bit complement of sixteen zeros is sixteen ones, which is the two's complement representation of -1.

NOT X = -(X+1)    The two's complement of any integer is the bit complement plus one.

## Functional Operators

A function is used in an expression to call a predetermined operation that is to be performed on an operand. Microsoft BASIC has "intrinsic" functions that reside in the system, such as SQR (square root) or SIN (sine). All of Microsoft BASIC's intrinsic functions are described in Chapter 4.

You may also define your own functions (known as "user-defined") with the DEF FN statement (see Chapter 3).

## String Operations

A string expression is an expression that contains string constant(s) or string variable(s), or a combination of both (with operators) that evaluates to a single value.

There are two classes of string operations: concatenation and functions.

## Concatenation

Combining two strings is called concatenation. The plus symbol (+) is the concatenation operator. For example,

```
Ok
10 A$="FILE" : B$="NAME"
20 PRINT A$ + B$
30 PRINT "NEW " + A$ + B$
RUN
FILENAME
NEW FILENAME
Ok
```

combines the string variables A$ and B$ to produce the value "FILENAME."

## String Functions

Strings may be compared using the same relational operators that are used with numbers:

$$= \qquad <> \qquad < \qquad > \qquad <= \qquad >=$$

A string function is the same as a numeric function except the result is a string value. String comparisons are made by taking one character at a time from each string and comparing the ASCII codes. If all the ASCII codes are the same, the strings are equal. If the ASCII codes differ, the lower code number precedes the higher. If, during string comparison, the end of one string is reached, the shorter string is said to be smaller. Leading and trailing blanks are significant. Examples:

```
"AA" < "AB"
"FILENAME" = "FILENAME"
"X&" > "X#"
"CL " > "CL"
"kg" > "KG"
"SMYTH" < "SMYTHE"
B$ < "9/12/78"          where B$ = "8/12/78"
```

Thus, string comparisons can be used to test string values or to alphabetize strings. All string constants used in comparison expressions must be enclosed in quotation marks.

## Input Editing

If an incorrect character is entered as a line is being typed, it can be deleted with Control-H or the ← (backspace) key. Both keys backspace over a character and erase it. Once a character has been deleted, simply continue typing the line as desired.

To delete a line that is in the process of being typed, type Control-U or press → (the retype key). A carriage return is executed automatically after the line is deleted.

To correct program lines in a program that is currently in memory, simply retype the line using the same line number. BASIC will automatically replace the old line with the new line.

To delete the entire program that currently resides in memory, enter the NEW command (see Chapter 3).

Microsoft BASIC has other sophisticated editing facilities that are part of the EDIT command. EDIT is discussed in Chapter 3.

# Error Messages

If BASIC detects an error that terminates program execution, an error message is printed. For a complete list of Microsoft BASIC error codes and error messages, see Appendix D.

# 3

## Microsoft BASIC Commands and Statements

# 3
# *Microsoft BASIC Commands and Statements*

Microsoft BASIC commands and statements are described in this chapter. Each description consists of the following components:

| | |
|---|---|
| Syntax | Shows the correct syntax for the instruction. |
| Purpose | Tells what the instruction is used for. |
| Remarks | Describes in detail how the instruction is used. |
| Example | Shows sample programs or program segments that demonstrate the use of the instruction. |

Syntax notation for all commands and statements is given in Chapter 1. Numeric and string arguments (where applicable) have been abbreviated as follows:

| | |
|---|---|
| X and Y | Represent any numeric expressions. |
| I and J | Represent integer expressions. |
| X$ and Y$ | Represent string expressions. |

If a floating-point value is supplied where an integer is required, BASIC will round the fractional portion and use the resulting integer.

## *AUTO*

| | |
|---|---|
| *Syntax* | AUTO [<line number>[,<increment>]] |
| *Purpose* | To generate a line number automatically after every carriage return. |
| *Remarks* | AUTO begins numbering at <line number> and increments each subsequent line number by <increment>. The default for both values is 10. If <line number> is followed by a comma but <increment> is not specified, the last increment specified in an AUTO command is assumed. |
| | If AUTO generates a line number that is already being used, an asterisk is printed after the number to warn the user that any input will replace the existing line. However, typing a carriage return immediately after the asterisk will save the line and generate the next line number. |
| | AUTO is terminated by typing Control-C. The line in which Control-C is typed is not saved. After Control-C is typed, BASIC returns to command level. |
| *Example* | AUTO 100,50   Generates line numbers 100, 150, 200 . . . . |
| | AUTO   Generates line numbers 10, 20, 30, 40 . . . . |

## *CALL*

| | |
|---|---|
| *Syntax* | CALL <variable name>[(<argument list>)] |
| *Purpose* | To call an assembly language subroutine. |

*Remarks*   The CALL statement is one way to transfer program flow to an external subroutine. (See also the USR function).

<variable name> contains an address that is the starting point in memory of the subroutine.

<variable name> may not be an array variable name.

<argument list> contains the arguments that are passed to the external subroutine. <argument list> may contain only variables.

*Example*   Ok
110 MYROUT=&HD000
120 CALL MYROUT(I,J,K)
.
.
.

# CHAIN

*Syntax*   CHAIN [MERGE] <filespec>[,[<line number exp>]
[,ALL][,DELETE<m-n>]]

*Purpose*   To call a program and pass variables to it from the current program.

*Remarks*   <filespec> contains the name of the program called. For example,

CHAIN"A:PROG1.BAS"

calls the BASIC program PROG1 from disk drive A. If no other options are included by the user, CHAIN will load the called program and execute it beginning at the first line.

The MERGE option of the CHAIN statement merges the called overlay into the currently running program. That is, the program lines of the overlay are inserted into the current program in sequential order beginning at the point specified by <line number exp>. The called program must be an ASCII file if it is to be merged. Example 1 in this section shows how the MERGE option is used.

*Note*

The CHAIN statement with MERGE option leaves the files open and preserves the current OPTION BASE setting.

When using the MERGE option, user-defined functions should be placed before any CHAIN MERGE statements in the program. Otherwise, the user-defined functions will be undefined after the merge is complete.

If you choose not to use MERGE, CHAIN will clear the effect of ON ERROR GOTO, disallow program continuation, reset all DATA pointers, and close all files. User-defined functions are preserved only if the corresponding DEF FN statements are not altered by MERGE. CHAIN without MERGE does not preserve variable types or user-defined functions for use by the chained program. That is, any DEFINT, DEFSNG, DEFDBL, DEFSTR, or DEF FN statements containing shared variables must be restated in the chained program.

<line number exp> is the line number (or an expression that evaluates to a line number) in the called program. It is the starting point for execution of the called program. If it is omitted, execution begins at the first line.

*Note*

<line number exp> is not affected by a RENUM command.

ALL is an option that allows all variables to pass from the current program to the overlay. If ALL is omitted, COMMON statements must be used to pass variables.

If the ALL option is used, every variable in the current program is passed to the overlay. If you do not use ALL, a COMMON statement must be used to pass variables to the overlay. With COMMON, you may specify the variables to be passed. Array variables may be used by appending parentheses () to the variable list. Note that the same variable cannot appear in more than one COMMON statement.

DELETE<m-n> is the option that deletes a range of lines in the original program after the overlay has been executed. m is the beginning line number and n is the last line number of the overlay range.

*Example*

```
10   REM THIS PROGRAM DEMONSTRATES
CHAINING WITH THE VARIABLES PASSED
20   REM IN COMMON. SAVE THIS PROGRAM
ON THE DISK AS "PROG1".
30   COMMON A$,B$
40   INPUT "ENTER A STRING OF NOT MORE
THAN 255 CHARACTERS. ",A$
50   B$ = " "
60   CHAIN "PROG2"
70   PRINT "HERE IS YOUR STRING BACK,
REVERSED!"
80   PRINT B$
90   END
```

```
10   REM SAVE THIS PROGRAM ON THE DISK
AS "PROG2"
20   COMMON A$, B$
30   FOR N% = 1 TO LEN(A$)
40      I% = LEN(A$) - N% + 1
50      B$ = B$ + MID$(A$, I%, 1)
60   NEXT N%
70   CHAIN "PROG1", 70
80   END
```

# CLEAR

**Syntax**    CLEAR [,[<expression1>][,<expression2>]]

**Purpose**    To set all numeric variables to zero, all string variables to null, and close all open files; and, optionally, to set the end of memory and the amount of stack space.

**Remarks**    <expression1> is a memory location which, if specified, sets the highest location available for use by BASIC.

<expression2> sets aside stack space for BASIC. The default is 256 bytes or one-eighth of the available memory, whichever is smaller.

BASIC allocates string space dynamically. An "Out of string space" error occurs only if there is no free memory left for BASIC to use.

**Examples**    CLEAR

CLEAR   ,32768

CLEAR   ,,2000

CLEAR   ,32768,2000

# CLOSE

**Syntax**    CLOSE[[#]<file number>[,[#]<file number ...>]]

**Purpose**    To conclude I/O to a disk file. CLOSE may be used either as a command or a statement.

*Remarks*      <file number> is the number under which the file was opened. A CLOSE with no arguments closes all open files.

The association between a particular file and file number terminates upon execution of a CLOSE. The file may then be reopened using the same or a different file number; likewise, that file number may now be reused to OPEN any file.

A CLOSE for a sequential output file writes the final buffer of output.

The END statement and the NEW command always CLOSE all disk files automatically. (STOP does not close disk files.)

*Examples*     CLOSE #1

CLOSE 2, 3

# COMMON

*Syntax*       COMMON <list of variables>

*Purpose*      To pass variables to a chained program.

*Remarks*      The COMMON statement is used in conjunction with the CHAIN statement. COMMON statements may appear anywhere in a program, though it is recommended that they appear at the beginning. The same variable cannot appear in more than one COMMON statement. Array variables are specified by appending "()" to the variable name. If all variables are to be passed, use CHAIN with the ALL option and omit the COMMON statement.

The <list of variables> may include any variable type, including array variables.

*Example*      100 COMMON A,B,C,D(),G$
110 CHAIN "PROG3",10
.
.
.

# CONT

*Syntax*       CONT

*Purpose*      To continue program execution after a Control-C has been typed or a STOP or END statement has been executed.

*Remarks*      Execution resumes at the point where the break occurred. If the break occurred after a prompt from an INPUT statement, execution continues with the reprinting of the prompt ("?" or prompt string).

CONT is usually used in conjunction with STOP for debugging. When execution is stopped, intermediate values may be examined and changed using direct mode statements. Execution may be resumed with CONT or a direct mode GOTO, which resumes execution at a specified line number. CONT may be used to continue execution after an error occurs.

CONT is invalid if the program has been edited during the break.

*Example*

```
Ok
10    INPUT A,B,C
20    K=A^2*5.3:L=B^3/.26
30    STOP
40    M=C*K+100:PRINT M
RUN
? 1,2,3
BREAK IN 30
Ok
PRINT L
 30.7692
Ok
CONT
 115.9
Ok
```

# DATA

*Syntax*    DATA <list of constants>

*Purpose*    To store the numeric and string constants that are accessed by the program's READ statement(s).

*Remarks*    DATA statements are nonexecutable and may be placed anywhere in the program. A DATA statement may contain as many constants as will fit on a line (separated by commas). Any number of DATA statements may be used in a program. The READ statements access the DATA statements in order (by line number) and the data contained therein may be thought of as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program.

<list of constants> may contain numeric constants in any format, i.e., fixed-point, floating-point or integer. (No numeric expressions are allowed in the list.) String constants in DATA statements must be surrounded by double quotation marks only if they contain commas, colons, or significant leading or trailing spaces. Otherwise, quotation marks are not needed.

The variable type (numeric or string) given in the READ statement must agree with the corresponding constant in the DATA statement.

DATA statements may be reread from the beginning by using the RESTORE statement.

*Example 1*

```
        .

        .

        .
80 FOR I=1 TO 10
90 READ A(I)
100 NEXT I
110 DATA 3 08,5.19,3.12,3.98,4.24
120 DATA 5.08,5.55,4.00,3.16,3.37

        .

        .

        .
```

This program segment reads the values from the DATA statements into the array A. After execution, the value of A(1) will be 3.08, and so on.

*Example 2*

```
Ok
10 PRINT "CITY", "STATE", " ZIP"
20 READ C$,S$,Z
30 DATA "DENVER,", COLORADO, 80211
40 PRINT C$,S$,Z
Ok
RUN
CITY            STATE           ZIP
DENVER,         COLORADO        80211
Ok
```

This program reads string and numeric data from the DATA statement in line 30.

# DEF FN

**Syntax**

DEF FN<name>[(<parameter list>)]=
<function definition>

**Purpose**

To define and name a function written by the user.

**Remarks**

<name> is any legal variable name. The name must be preceded by FN and becomes the name of the function.

is comprised of those variable names in the function definition that are to be replaced when the function is called. The items in the list are separated by commas.

The variables in the parameter list represent, on a one-to-one basis, the argument variables or values that will be given in the function call.

<function definition> is an expression that performs the operation of the function. It is limited to one line. Variable names that appear in this expression serve only to define the function; they do not affect program variables that have the same name.

A variable name used in a function definition may or may not appear in the parameter list. If it does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the variable is used.

User-defined functions may be numeric or string. If a type is specified in the function name, the value of the expression is forced to that type before it is returned to the calling statement. If a type is specified in the function name and the argument type does not match, a "Type mismatch" error occurs.

A DEF FN statement must be executed before the function it defines may be called. If a function is called before it has been defined, an "Undefined user function" error occurs. DEF FN is illegal in direct mode.

**Example**

```
410 DEF FNAB(X,Y)=X^3/Y^2
420 T=FNAB(I,J)
```

Line 410 defines the function FNAB. The function is called in line 420.

# DEFINT/SNG/DBL/STR

**Syntax**

DEF<type> <range(s) of letters>

**Purpose**

To declare variable types as integer, single precision, double precision, or string.

**Remarks**

<type> is a variable type (INT, SNG, DBL, or STR) and <range of letters> is the variable name(s).

A DEF statement declares that the variable names beginning with the letter(s) specified will assume that variable type. However, a type declaration character always takes precedence over a DEF statement in the typing of a variable.

If no type declaration statements are encountered, BASIC assumes that all variables without declaration characters are single precision variables.

*Examples*    10 DEFDBL L-P    All variables beginning with the letters L, M, N, O, and P will be double precision  variables.

10 DEFSTR A    All variables beginning with the letter A will be string  variables.

10 DEFINT I-N,W-Z

All variables beginning with the letters I, J, K, L, M, N, W, X, Y, Z will be integer variables.

# DEF USR

*Syntax*    DEF USR[<digit>]=<integer expression>

*Purpose*    To specify the starting address of an assembly language subroutine.

*Remarks*    <digit> may be any digit from 0 to 9. The digit corresponds to the number of the USR routine whose address is being specified. If <digit> is omitted, DEF USR0 is assumed.

The value of <integer expression> is the starting address of the USR routine (see Appendix C, "BASIC Assembly Language Subroutines").

Any number of DEF USR statements may appear in a program to redefine subroutine starting addresses, thus allowing access to as many subroutines as necessary.

*Example*

.

.

.

200 DEF USR0=24000
210 X=USR0(Y^2/2.89)

.

.

.

# DELETE

*Syntax*    DELETE[<line number>][-<line number>]

*Purpose*    To delete program lines.

*Remarks*    BASIC always returns to command level after DELETE is executed. If <line number> does not exist, an "Illegal function call" error occurs.

*Examples*    DELETE 40    Deletes line 40.

DELETE 40-100    Deletes lines 40 through 100, inclusive.

DELETE-40    Deletes all lines up to and including line 40.

# DIM

*Syntax*    DIM <list of subscripted variables>

*Purpose*    To specify the maximum values for array variable subscripts and allocate storage accordingly.

*Remarks*    If an array variable name is used without a DIM statement, the maximum value of its subscript(s) is

assumed to be 10. If a subscript is used that is greater than the maximum specified, a "Subscript out of range" error occurs. The minimum value for a subscript is always zero, unless otherwise specified with the OPTION BASE statement.

The DIM statement sets all the elements of the specified arrays to an initial value of zero.

*Example*

```
Ok
10 DIM A(20)
20 FOR I=0 TO 20
30 READ A(I)
40 NEXT I
   .
   .
   .
```

# EDIT

*Syntax*

EDIT <line number>

*Purpose*

To enter the edit mode at a specified line.

*Remarks*

In edit mode, it is possible to edit portions of a line without retyping the entire line. Upon entering edit mode, BASIC types the line number of the line to be edited, then it types a space and waits for an edit mode subcommand.

*Edit mode subcommands*

Edit mode subcommands are used to move the cursor or to insert, delete, replace, or search for text within a line. The subcommands are not echoed. Most of the edit mode subcommands may be preceded by an integer which causes the command to be executed that number of times. When a preceding integer is not specified, it is assumed to be 1.

Edit mode subcommands may be categorized according to the following functions:

1. Moving the cursor
2. Inserting text
3. Deleting text
4. Finding text
5. Replacing text
6. Ending and restarting edit mode
7. Syntax errors

Descriptions of each category follow.

### Note

In the descriptions that follow, <ch> represents any character, <text> represents a string of characters of arbitrary length, *i* represents an optional integer (the default is 1), and $ represents the <ESCAPE> key.

Moving the Cursor

Space

Use the space bar to move the cursor to the right. *i*Space moves the cursor *i* spaces to the right. Characters are printed as you space over them.

Control-H

In edit mode, *i*Control-H moves the cursor *i* spaces to the left. Characters are printed as you backspace over them.

←

← or backspace moves the cursor to the left. The cursor moves over the characters already printed, but does not delete them.

## Inserting Text

I    I<text>$ inserts <text> at the current cursor position. The inserted characters are printed on the screen. To terminate insertion, type <ESCAPE>. If <RETURN> is typed during an Insert command, the effect is the same as typing <ESCAPE> and then <RETURN>. During an Insert command, Control-H, < ← >, or the Underscore key may be used to delete characters to the left of the cursor. Control-H will move the cursor over the characters as you backspace over them. <SHIFT, Control-\ > and Underscore (when pressed simultaneously) will print an Underscore for each character you delete. If an attempt is made to insert a character that will make the line longer than 255 characters, an audio beep (Control-G) sounds and the character is not printed.

X    The X subcommand is used to extend the line. X moves the cursor to the end of the line, enters the insert mode, and allows insertion of text as if an insert command had been given. When you are finished extending the line, type <ESCAPE> or <RETURN>.

## Deleting Text

D    iD deletes i characters to the right of the cursor. The deleted characters are echoed between backslashes, and the cursor is positioned to the right of the last character deleted. If there are fewer than i characters to the right of the cursor, iD deletes the remainder of the line.

H    H deletes all characters to the right of the cursor and then automatically enters the insert submode. H is useful for replacing statements at the end of a line.

## Finding Text

S    The subcommand iS<ch> searches for the ith occurrence of <ch> and positions the cursor before it. The character at the current cursor position is not included in the search. If <ch> is not found, the cursor will stop at the end of the line. All characters passed over during the search are printed.

K    The subcommand iK<ch> is similar to iS<ch>, except all the characters passed over in the search are deleted. The cursor is positioned before <ch>, and the deleted characters are enclosed in backslashes.

## Replacing Text

C    The subcommand C<ch> changes the next character to <ch>. If you wish to change the next i characters, use the subcommand iC, followed by i characters. After the ith new character is typed, you exit the change submode and return to edit mode.

## Ending and Restarting Edit Mode

Control-A    To enter edit mode on the line you are currently typing, type Control-A. BASIC responds with a return, an exclamation point (!), and a space. The cursor will be positioned at the first character in the line. Proceed by typing an edit mode subcommand.

***Note***

If you have just entered a line, and wish to go back and edit it, the command "EDIT ." will enter edit mode at the current line. (The line number symbol "." always refers to the current line.)

<RETURN>    Typing <RETURN> prints the remainder of the line, saves the changes you made, and exits edit mode.

E    The E subcommand has the same effect as <RETURN>, except the remainder of the line is not printed.

Q    The Q subcommand returns to BASIC command level without saving any of the changes that were made to the line during edit mode.

L    The L subcommand lists the remainder of the line (saving any changes made so far) and repositions the cursor at the beginning of the line, while still in edit mode. L is usually used to list the line when first entering edit mode.

A    The A subcommand lets you begin editing a line over again. It restores the original line and repositions the cursor at the beginning.

***Note***

If BASIC receives an unrecognizable command or illegal character while in edit mode, it sounds a beep (Control-G) and the command or character is ignored.

Syntax Errors

When a syntax error is encountered during execution of a program, BASIC automatically enters edit mode at the line that caused the error.

For example:

```
10 K = 2(4)
RUN
?Syntax error in 10
10
```

When you finish editing the line and press <RETURN> (or the E subcommand), BASIC reinserts the line, which causes all variable values to be lost. To preserve the variable values for examination, first exit edit mode with the Q subcommand. BASIC will return to command level, and all variable values will be preserved.

## END

| | |
|---|---|
| ***Syntax*** | END |
| ***Purpose*** | To terminate program execution, close all files and return to command level. |
| ***Remarks*** | END statements may be placed anywhere in the program to terminate execution. Unlike the STOP statement, END does not cause a "BREAK" message to be printed. An END statement at the end of a program is optional. BASIC always returns to command level after an END is executed. |
| ***Example*** | 520 IF K>1000 THEN END ELSE GOTO 20 |

# ERASE

| | |
|---|---|
| **Syntax** | ERASE <array variable> [<array variable>...] |
| **Purpose** | To eliminate arrays from a program. |
| **Remarks** | Arrays may be redimensioned after they are erased, or the previously allocated array space in memory may be used for other purposes. If an attempt is made to redimension an array without first erasing it, a "Redimensioned array" error occurs. |
| **Example** | 10 DIM B(5) |

```
.
.
.
450 ERASE A,B
460 DIM B(99)
.
.
.
```

# ERROR

| | |
|---|---|
| **Syntax** | ERROR <integer expression> |
| **Purpose** | (1) To simulate the occurrence of a BASIC error; or, (2) to allow error codes to be defined by the user. |
| **Remarks** | <integer expression> must be a value between 0 and 255. If the value of <integer expression> equals an error code already in use by BASIC (see Appendix D), the ERROR statement will simulate the occurrence of that error, and the corresponding error message will be printed. (See Example 1.) |

To define your own error code, use a value that is greater than any used by the Microsoft BASIC error codes. (It is preferable to use the highest available values, so compatibility may be maintained when more error codes are added to Microsoft BASIC.) This user-defined error code may then be conveniently handled in an error trap routine. (See Example 2.)

If an ERROR statement specifies a code for which no error message has been defined, BASIC responds with the message "Unprintable error." Execution of an ERROR statement for which there is no error trap routine causes an error message to be printed and execution to halt.

| | |
|---|---|
| **Example 1** | Ok<br>10 S = 10<br>20 T = 5<br>30 ERROR S + T<br>40 END<br>Ok<br>RUN<br>String too long in line 30 |

Or, in direct mode:

```
Ok
ERROR 15          (You type this line.)
String too long   (BASIC types this line.)
Ok
```

| | |
|---|---|
| **Example 2** | . |

```
.
.
.
110 ON ERROR GOTO 400
120 INPUT "WHAT IS YOUR BET";B
130 IF B > 5000 THEN ERROR 210
.
.
.
```

```
400 IF ERR = 210 THEN PRINT
"HOUSE LIMIT IS $5000"
410 IF ERL = 130 THEN RESUME 120
    .
    .
    .
```

**Note**

Refer to the section entitled "ERR and ERL Variables" for more information on ERR and ERL.

# FIELD

| | |
|---|---|
| **Syntax** | FIELD[#]<file number>, <field width> AS <string variable> . . . |
| **Purpose** | To allocate space for variables in a random file buffer. |
| **Remarks** | A FIELD statement must be executed to get data out of a random buffer after a GET statement or to enter data before a PUT statement. |

The FIELD statement contains three user entries. <file number> is the number under which the file was opened. <field width> is the number of characters to be allocated to <string variable>. <string variable> is a string variable that will be used for random file access.

The total number of bytes allocated in a FIELD statement must not exceed the record length that was specified when the file was opened (see the OPEN command in this chapter). Otherwise, a "Field overflow" error occurs. (The default record length is 128.)

Any number of FIELD statements may be executed for the same file, and all FIELD statements that have been executed are in effect at the same time.

You may not use a fielded variable name in an INPUT or LET statement. Once a variable name is fielded, it points to the correct place in the random file buffer. If a subsequent INPUT or LET statement with that variable name is executed, the variable's pointer is moved to string space.

| | |
|---|---|
| **Example 1** | FIELD 1, 20 AS N$, 10 AS ID$, 40 AS ADD$ |

Allocates the first 20 positions (bytes) in the random file buffer to the string variable N$, the next 10 positions to ID$, and the next 40 positions to ADD$. FIELD does NOT place any data in the random file buffer. (See LSET/RSET and GET.)

**Example 2**
```
10 OPEN "A:DEPT" AS 1
20 FIELD #1, 30 AS DEPTNAME$, 30 AS LOCA$,
30 GET #1
40 D$ = DEPTNAME$
50 PRINT D$, DEPTNAME$, LOCA$
```

# FILES

| | |
|---|---|
| **Syntax** | FILES[<filespec>] |
| **Purpose** | To print the names of files residing on the current disk. |
| **Remarks** | <filespec> is a string formula which may contain question marks (?) to match any character in the filename or extension. An asterisk (*) as the first character of the filename or extension will match any file or any extension. |

If a disk drive is specified as part of the <filespec>, then files under the specified filename in that disk drive are listed. Otherwise, the current or default drive is used.

| | |
|---|---|
| *Examples* | FILES |
| | FILES "*.BAS" |
| | FILES "B:*.*" |
| | FILES "TEST?.BAS" |

# FOR . . . NEXT

| | |
|---|---|
| *Syntax* | FOR <variable>=<x> TO <y> [STEP <z>] |
| | . |
| | . |
| | . |
| | NEXT [<variable>][,<variable> . . . ] |

*Purpose*   To allow a series of instructions to be performed in a loop a given number of times.

*Remarks*   <variable> is used as a counter and <x>, <y>, and <z> are numeric expressions. The first numeric expression, <x>, is the initial value of the counter. The second numeric expression, <y>, is the final value of the counter.

The program lines following the FOR statement are executed until the NEXT statement is encountered. Then the counter is incremented by the amount specified by STEP. A check is performed to see if the value of the counter is now greater than the final value of <y>. If it is not greater, BASIC branches back to the statement after the FOR statement and the process is repeated. If it is greater, execution continues with the statement following the NEXT statement. This is a FOR . . . NEXT loop. If STEP is not specified, the increment is assumed to be one. If STEP is negative, the final value of the counter is set to be less than the initial value. The counter is decremented each time through the loop, and the loop is executed until the counter is less than the final value.

The body of the loop is skipped if the initial value of the loop times the sign of the step exceeds the final value times the sign of the step.

*Nested Loops*

FOR . . . NEXT loops may be nested; that is, a FOR . . . NEXT loop may be placed within the context of another FOR . . . NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before that for the outside loop. If nested loops have the same end point, a single NEXT statement may be used for all of them.

The variable(s) in the NEXT statement may be omitted, in which case the NEXT statement will match the most recent FOR statement. If a NEXT statement is encountered before its corresponding FOR statement, a "NEXT without FOR" error message is issued and execution is terminated.

| | |
|---|---|
| *Example 1* | Ok |
| | 10 K=10 |
| | 20 FOR I=1 TO K STEP 2 |
| | 30 PRINT I; |
| | 40 K=K+10 |
| | 50 PRINT K |
| | 60 NEXT |
| | RUN |
| |  1  20 |
| |  3  30 |
| |  5  40 |
| |  7  50 |
| |  9  60 |
| | Ok |
| *Example 2* | Ok |
| | 10 J=0 |
| | 20 FOR I=1 TO J |
| | 30 PRINT I |
| | 40 NEXT I |

In this example, the loop does not execute because the initial value of the loop exceeds the final value.

*Example 3*

```
Ok
10 I=5
20 FOR I=1 TO I+5
30 PRINT I;
40 NEXT
RUN
 1 2 3 4 5 6 7 8 9 10
Ok
```

In this example, the loop executes ten times. The final value for the loop variable is always set before the initial value is set.

# GET

*Syntax*    GET [#]<file number>[,<record number>]

*Purpose*    To read a record from a random disk file into a random buffer.

*Remarks*    <file number> is the number under which the file was opened and <record number> is the number of the record to be read. The range is 1 to 32767.

If <record number> is omitted, the next record (after the last GET) is read into the buffer. The largest possible record number is 32767.

### Note

After a GET statement, INPUT# and LINE INPUT# may be executed to read characters from the random file buffer.

*Example*

```
Ok
10 OPEN "B: VENDOR AS #1
20 FIELD #1, 20 AS VENDNAME$, 30 AS ADDR$,
    35 AS CITY$
30 GET #1
40 PRINT VENDNAME$, ADDR$, CITY$
```

# GOSUB...RETURN

*Syntax*    GOSUB <line number>
.
.
.
RETURN

*Purpose*    To branch to and return from a subroutine.

*Remarks*    <line number> is the first line of the subroutine. A subroutine may be called any number of times in a program, and a subroutine may be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

The RETURN statement(s) in a subroutine cause BASIC to branch back to the statement following the most recent GOSUB statement. A subroutine may contain more than one RETURN statement, should logic dictate a return at different points in the subroutine. Subroutines may appear anywhere in the program, but it is recommended that the subroutine be readily distinguishable from the main program.

To prevent inadvertent entry into the subroutine, the GOSUB statement may be preceded by a STOP, END, or GOTO statement that directs program control around the subroutine.

To prevent stack overflow, a subroutine called by a GOSUB statement must always exit through a RETURN statement.

You may use an ON . . . GOSUB statement to branch to different subroutines based on the result of an expression.

*Example*

```
Ok
10 GOSUB 40
20 PRINT "BACK FROM SUBROUTINE"
30 END
40 PRINT "SUBROUTINE";
50 PRINT "IN";
60 PRINT "PROGRESS"
70 RETURN
RUN
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
Ok
```

# GOTO

*Syntax*      GOTO <line number>

*Purpose*     To branch unconditionally out of the normal program sequence to a specified line number.

*Remarks*    If <line number> is an executable statement, that statement and those following are executed. If it is a nonexecutable statement, execution proceeds at the first executable statement encountered after <line number>.

*Example*

```
Ok
10 READ R
20 PRINT "R =";R,
30 A = 3.14*R^2
40 PRINT "AREA =";A
50 GOTO 10
60 DATA 5,7,12
Ok
RUN
R = 5        AREA = 78.5
R = 7        AREA = 153.86
R = 12       AREA = 452.16
?Out of data in 10
Ok
```

# IF . . . THEN [. . . ELSE] and IF . . . GOTO

*Syntax*     IF <expression> THEN <clause>
            [ELSE <clause>]

*Syntax*     IF <expression> GOTO <line number>
            [ELSE <clause>]

*Purpose*    To make a decision regarding program flow based on the result returned by an expression.

*Remarks*    <expression> is a unique expression setting the conditions for the IF statement to make a decision on which program path to follow. <clause> may be a BASIC statement or statements, or a line number to branch to.

If the result of <expression> is not zero, the THEN or GOTO clause is executed. THEN may be followed by either a line number for branching or by one or more statements to be executed. GOTO is always followed by a line number. If the result of

<expression> is zero, the THEN or GOTO clause is ignored and the ELSE clause, if present, is executed. Execution continues with the next executable statement. A comma is allowed before THEN.

*Nesting of IF Statements*

IF . . . THEN . . . ELSE statements may be nested. Nesting is limited only by the length of the line. For example,

IF X>Y THEN PRINT "GREATER" ELSE IF Y>X
THEN PRINT "LESS THAN" ELSE PRINT "EQUAL"

is a legal statement. If the statement does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN. For example,

IF A=B THEN IF B=C THEN PRINT "A=C"
ELSE PRINT "A<>C"

will not print "A<>C" when A<>B.

If an IF . . . THEN statement is followed by a line number in the direct mode, an "Undefined line" error results unless a statement with the specified line number had previously been entered in indirect mode.

### Note

When using IF to test equality for a value that is the result of a floating point computation, remember that the internal representation of the value may not be exact. Therefore, the test should be against the range over which the accuracy of the value may vary. For example, to test a

computed variable A against the value 1.0, use:

IF ABS (A-1.0)<1.0E-6 THEN ...

This test returns true if the value of A is 1.0 with a relative error of less than 1.0E-6.

*Example 1*    200 IF I THEN GET#1,I

This statement GETs record number I if I is not zero.

*Example 2*    100 IF(I<20)*(I>10) THEN DB=1979-1:GOTO 300
110 PRINT "OUT OF RANGE"

.
.
.

In this example, a test determines if I is greater than 10 and less than 20. If I is in this range, DB is calculated and execution branches to line 300. If I is not in this range, execution continues with line 110.

*Example 3*    210 IF IOFLAG THEN PRINT A$ ELSE LPRINT A$

This statement causes printed output to go either to the screen or the line printer, depending on the value of a variable (IOFLAG). If IOFLAG is zero, output goes to the line printer; otherwise, output goes to the screen.

# INPUT

*Syntax*    INPUT[;][<"prompt string">;]<variable list>

*Purpose*    To allow input from the keyboard during program execution.

**Remarks**    When an INPUT statement is encountered, program
execution pauses and a question mark is displayed
to indicate the program is waiting for data. If
<"prompt string"> is included, the string is
displayed before the question mark. The required
data is then entered at the keyboard.

A comma may be used instead of a semicolon after
the prompt string to suppress the question mark. For
example, the statement INPUT "ENTER
BIRTHDATE",B$ will display the prompt with no
question mark.

If INPUT is immediately followed by a semicolon,
then the <RETURN> typed by the user to input data
does not echo a carriage return/linefeed sequence.

The data that is entered is assigned to the
variable(s) given in <variable list>. The number of
data items supplied must be the same as the
number of variables in the list. Data items are
separated by commas.

The variable names in the list may be numeric or
string variable names (including subscripted
variables). The type of each data item that is input
must agree with the type specified by the variable
name. (Strings input to an INPUT statement need
not be surrounded by quotation marks.)

Responding to INPUT with too many or too few
items, or with the wrong type of value (numeric
instead of string, etc.) causes the messsage "?Redo
from start" to be printed. No assignment of input
values is made until an acceptable response is
given.

**Examples**    Ok
10 INPUT X
20 PRINT X "SQUARED IS" X^2
30 END
RUN
?  5     (The 5 was typed in by the user in response
            to the question mark.)
 5  SQUARED IS 25
Ok

Ok
10 PI=3.14
20 INPUT "WHAT IS THE RADIUS";R
30 A=PI*R^2
40 PRINT "THE AREA OF THE CIRCLE IS";A
50 PRINT
60 GOTO 20
Ok
RUN
WHAT IS THE RADIUS? 7.4      (User types 7.4)
THE AREA OF THE CIRCLE IS 171.946

WHAT IS THE RADIUS?

# INPUT#

**Syntax**    INPUT #<file number>,<variable list>

**Purpose**    To read data items from a sequential disk file and
assign them to program variables.

**Remarks**    <file number> is the number used when the file was
opened for input. <variable list> contains the
variable names that will be assigned to the items in
the file. (The variable type must match the type
specified by the variable name.)

The data items in the file should appear just as they would if data were being typed in response to an INPUT statement. Unlike INPUT, no question mark is printed with INPUT#.

With numeric values, leading spaces, carriage returns and line feeds are ignored. The first character encountered that is not a space, carriage return or line feed is assumed to be the start of a number. The number terminates on a space, carriage return, line feed, or comma.

If BASIC is scanning the sequential data file for a string item, leading spaces, carriage returns, and line feeds are also ignored. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of a string item. If this first character is a quotation mark ("), the string item will consist of all characters read between the first quotation mark and the second. Thus, a quoted string may not contain a quotation mark as a character. If the first character of the string is not a quotation mark, the string is an unquoted string, and will terminate on a comma, carriage return, or line feed (or after 255 characters have been read). If end of file is reached when a numeric or string item is being INPUT, the item is terminated.

*Example*

```
Ok
10 OPEN "I",#1,"DATA"
20 INPUT#1,N$,D$,H$
30 IF RIGHT$(H$,2)="78" THEN PRINT N$
40 GOTO 20
RUN
EBENEEZER SCROOGE
SUPER MANN
Input past end in 20
Ok
```

# KILL

*Syntax*      KILL <filespec>

*Purpose*     To delete a file from disk.

*Remarks*     KILL is used for all types of disk files: program files, random data files, and sequential data files.

              If a KILL statement is given for a file that is currently OPEN, a "File already open" error occurs.

*Example*
```
Ok
10 ON ERROR GOTO 2000
20 OPEN "I",#1,"NAMES"
30 REM IF FILE EXISTS, WRITE IT TO "COPY"
40 OPEN "O",#2,"COPY"
50 IF EOF(1) THEN 90
60 LINE INPUT#1,A$
70 PRINT#2,A$
80 GOTO 50
90 CLOSE #1
100 KILL "NAMES"
110 REM ADD NEW ENTRIES TO FILE
120 INPUT "NAME";N$
130 IF N$="" THEN 200 'CARRIAGE RETURN EXITS
INPUT LOOP
140 LINE INPUT "ADDRESS? ";A$
150 LINE INPUT "BIRTHDAY? ";B$
160 PRINT#2,N$
170 PRINT#2,A$
180 PRINT#2,B$
190 PRINT:GOTO 120
200 CLOSE
205 REM CHANGE FILENAME BACK TO "NAMES"
210 NAME "COPY" AS "NAMES"
2000 IF ERR=53 AND ERL=20 THEN OPEN
"O",#2,"COPY":RESUME 120
2010 ON ERROR GOTO 0
```

See also Appendix B, "Microsoft BASIC Disk I/O."

# LET

**Syntax**    [LET] <variable>=<expression>

**Purpose**    To assign the value of an expression to a variable.

**Remarks**    Notice the word LET is optional; i.e., the equal sign is sufficient when assigning an expression to a variable name.

Attempting to assign a numeric value to a string variable or a string value to a numeric variable will result in a "Type mismatch" error.

**Example**
```
110 LET D=12
120 LET E=12^2
130 LET F=12^4
140 LET SUM=D+E+F
   .
   .
   .
```

or

```
110 D=12
120 E=12^2
130 F=12^4
140 SUM=D+E+F
   .
   .
   .
```

# LINE INPUT

**Syntax**    LINE INPUT[;][<"prompt string">;]
              <string variable>

**Purpose**    To input an entire line (up to 254 characters) to a string variable, without the use of delimiters.

**Remarks**    The <"prompt string"> is a string literal that is displayed on the screen before input is accepted. A question mark is not printed unless it is part of the prompt string.

<string variable> is the input. All input from the end of the prompt to the <RETURN> is assigned to <string variable>. However, if a line feed/carriage return sequence (this order only) is encountered, both characters are echoed, but the carriage return is ignored, the line feed is put into <string variable>, and data input continues.

If LINE INPUT is immediately followed by a semicolon, then the carriage return typed by the user to end the input line does not echo a carriage return/line feed sequence at the keyboard.

A LINE INPUT may be escaped by typing Control-C. BASIC will return to command level and type Ok. Typing CONT resumes execution at the LINE INPUT.

**Example**    See the example in the following section (LINE INPUT#).

# LINE INPUT #

**Syntax**    LINE INPUT#<file number>,<string variable>

**Purpose**    To read an entire line (up to 254 characters), without delimiters, from a sequential disk data file to a string variable.

**Remarks**   <file number> is the number under which the file was opened and <string variable> is the variable name to which the line will be assigned. LINE INPUT# reads all characters in the sequential file up to a <RETURN>. It then skips over the line feed/carriage return sequence. The next LINE INPUT# reads all characters up to the next <RETURN>. (If a line feed/carriage return sequence is encountered, it is preserved.)

LINE INPUT# is especially useful if each line of a data file has been broken into fields, or if a BASIC program saved in ASCII mode is being read as data by another program.

```
Ok
10 OPEN "O",1,"LIST"
20 LINE INPUT "CUSTOMER INFORMATION? ";C$
30 PRINT #1, C$
40 CLOSE 1
50 OPEN "I",1,"LIST"
60 LINE INPUT #1, C$
70 PRINT C$
80 CLOSE 1
RUN
CUSTOMER INFORMATION? LINDA JONES
234,4 MEMPHIS
LINDA JONES 234,4 MEMPHIS
Ok
```

# LIST

**Syntax 1**   LIST [<line number>]

**Syntax 2**   LIST [<line number>[-[<line number>]]]

**Purpose**   To list all or part of the program currently in memory on the screen.

**Remarks**   BASIC always returns to command level after LIST is executed.

Syntax format 1

If <line number> is omitted, the program is listed beginning at the lowest line number. (Listing is terminated either by the end of the program or by typing Control-C.) If <line number> is included, only the specified line is listed. Control-S suspends a listing. Pressing Control-S again (or Control-Q or any other key) allows the listing to continue.

Syntax format 2

This format allows the following options:

1.   If only the first number is specified, that line and all subsequent lines are listed.

2.   If only the second number is specified, all lines from the beginning of the program through that line are listed.

3.   If both numbers are specified, the entire range is listed.

**Examples**   Syntax format 1

| | |
|---|---|
| LIST | Lists the program currently in memory. |
| LIST 500 | Lists line 500. |

Syntax format 2

| | |
|---|---|
| LIST 150- | Lists all lines from 150 to the end. |
| LIST - 1000 | Lists all lines from the lowest number through 1000. |
| LIST 150 - 1000 | Lists lines 150 through 1000, inclusive. |

## LLIST

**Syntax**          LLIST [<line number>[-[<line number>]]]

**Purpose**         To list all or part of the program currently in memory to the line printer.

**Remarks**         BASIC always returns to command level after an LLIST is executed. The options for LLIST are the same as for LIST, Syntax 2.

<line number> is a valid line number in the range 0 to 65529.

LLIST assumes a 132 character wide printer.

**Examples**        LLIST 150 -        Lists all lines from 150 to the end.

LLIST -1000        Lists all lines from the lowest number through 1000.

LLIST 150 - 1000   Lists lines 150 through 1000, inclusive.

## LOAD

**Syntax**          LOAD <filespec>[,R]

**Purpose**         To load a file from disk into memory.

**Remarks**         <filespec> includes the name and extension for the file saved. With CP/M, the default extension .BAS is supplied.

LOAD closes all open files and deletes all variables and program lines currently residing in memory before it loads the designated program. However, if the "R" option is used with LOAD, the program is RUN after it is loaded, and all open data files are kept open. Thus, LOAD with the "R" option may be used to chain several programs (or segments of the same program). Information may be passed between the programs using disk data files.

**Example**         LOAD "STRTRK",R

## LPRINT and LPRINT USING

**Syntax**          LPRINT [<list of expressions>]

**Syntax**          LPRINT USING <string exp>;<list of expressions>

**Purpose**         To print data at the line printer.

**Remarks**         Same as PRINT and PRINT USING, except output goes to the line printer.

LPRINT assumes a 132 character wide printer.

## LSET and RSET

**Syntax**          LSET <string variable> = <string expression>

**Syntax**          RSET <string variable> = <string expression>

**Purpose**         To move data from memory to a random file buffer (in preparation for a PUT statement).

*Remarks*   If <string expression> requires fewer bytes than were fielded to <string variable>, LSET left justifies the string in the field, and RSET right justifies the string. (Spaces are used to pad the extra positions.) If the string is too long for the field, characters are dropped from the right. Numeric values must be converted to strings before they are LSET or RSET. See the MKI$, MKS$, MKD$ functions in Chapter 4.

### Note

LSET or RSET may also be used with a nonfielded string variable to left justify or right justify a string in a given field. For example, the program lines

```
110 A$=SPACE$(20)
120 RSET A$=N$
```

right justify the string N$ in a 20-character field. This can be very handy for formatting printed output.

*Examples*   
```
150 LSET A$=MKS$(AMT)
160 LSET D$=DESC$
```

See also Program 6 in Appendix B, "Microsoft BASIC Disk I/O."

# MERGE

*Syntax*   MERGE <filespec>

*Purpose*   To merge a specified ASCII disk file into the program currently in memory.

*Remarks*   <filespec> is the filename and extension of the file saved. CP/M will append a default filename extension of .BAS if one was not supplied in the

SAVE command. Refer to "File Naming Conventions" in Chapter 2 for more information about possible filename extensions under CP/M. The file must be saved in ASCII format. (If not, a "Bad file mode" error occurs.)

If any lines in the disk file have the same line numbers as lines in the program in memory, the lines from the file on disk will replace the corresponding lines in memory. (Merging may be thought of as "inserting" the program lines on disk into the program in memory.)

BASIC always returns to command level after executing MERGE.

*Example*   MERGE "A:CATPRO"

See also Example 1 for the CHAIN statement in this chapter.

# MID$

*Syntax*   MID$(<string exp1>,n[,m])=<string exp2>

*Purpose*   To replace a portion of one string with another string.

*Remarks*   n and m are integer expressions and <string exp1> and <string exp2> are string expressions.

The characters in <string exp1>, beginning at position n, are replaced by the characters in <string exp2>. The optional m refers to the number of characters from <string exp2> that will be used in the replacement. If m is omitted, all of <string exp2> is used. However, regardless of whether m is omitted or included, the replacement of characters

never goes beyond the original length of <string exp1>.

MID$ is also a function that returns a substring of a given string (see Chapter 4).

*Example*

```
Ok
10 A$="KANSAS CITY, MO"
20 MID$(A$,14)="KS"
30 PRINT A$
Ok
RUN
KANSAS CITY, KS
```

# NAME

*Syntax*        NAME <filespec> AS <new filename>

*Purpose*       To change the name of a disk file.

*Remarks*       <filespec> is a file specification as outlined under "File Naming Conventions" in Chapter 2. <new filename> is the new filename. It must be a valid filename as outlined in the same section.

<filespec> must exist and <new filename> must not exist; otherwise, an error will result. If the device name is omitted, the current drive is assumed. After NAME is executed, the file exists on the same disk, in the same area of disk space, with the new name.

*Example*

```
Ok
NAME "A:ACCTS" AS "LEDGER"
Ok
```

In this example, the disk file that was formerly named ACCTS in drive A will now be named LEDGER.

# NEW

*Syntax*        NEW

*Purpose*       To delete the program currently in memory and clear all variables.

*Remarks*       NEW is entered at command level to clear memory before entering a new program. BASIC always returns to command level after a NEW is executed.

# ON ERROR GOTO

*Syntax*        ON ERROR GOTO <line number>

*Purpose*       To enable error trapping and specify the first line of the error handling subroutine.

*Remarks*       Once error trapping has been enabled, all errors detected, including direct mode errors (e.g., syntax errors), will generate a jump to the specified error handling subroutine. If <line number> does not exist, an "Undefined line" error results. To disable error trapping, execute an ON ERROR GOTO 0. Subsequent errors generate an error message and halt execution. An ON ERROR GOTO 0 statement that appears in an error trapping subroutine causes BASIC to stop and print the error message for the error that caused the trap. It is recommended that all error trapping subroutines execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

*Note*

If an error occurs during execution of an error handling subroutine, the BASIC error

message is printed and execution terminates. Error trapping does not occur within the error handling subroutine.

*Example*      10 ON ERROR GOTO 1000

# ON . . . GOSUB and ON . . . GOTO

*Syntax*      ON <expression> GOTO <list of line numbers>

*Syntax*      ON <expression> GOSUB <list of line numbers>

*Purpose*     To branch to one of several specified line numbers, depending on the value returned when an expression is evaluated.

*Remarks*     The value of <expression> determines which line number in the list will be used for branching. For example, if the value is three, the third line number in the list will be the destination of the branch. (If the value is a non-integer, the fractional portion is rounded.)

In the ON . . . GOSUB statement, each line number in the list must be the first line number of a subroutine.

If the value of <expression> is zero or greater than the number of items in the list (but less than or equal to 255), BASIC continues with the next executable statement. If the value of <expression> is negative or greater than 255, an "Illegal function call" error occurs.

*Example*     100 ON L-1 GOTO 150,300,320,390

# OPEN

*Syntax*      OPEN <mode>,[#]<file number>,<filespec>, [<reclen>]

*Purpose*     To allow I/O to a disk file.

*Remarks*     <mode> is a string expression whose first character is one of the following:

O         specifies sequential output mode

I         specifies sequential input mode

R         specifies random input/output mode

<file number> is an integer expression whose value is between one and fifteen. The number is then associated with the file for as long as it is OPEN and is used to refer other disk I/O statements to the file.

<filespec> is a string expression for file specification which contains a name that conforms to CP/M's rules for disk filenames.

<reclen> is an integer expression which, if included, sets the record length for random files. <reclen> is not valid for sequential files. The default record length is 128 bytes. To use OPEN with record lengths longer than 128 bytes, see "Initialization," in Chapter 2.

A disk file must be opened before any disk I/O operation can be performed on that file. OPEN allocates a buffer for I/O to the file and determines the mode of access that will be used with the buffer.

*Note*

A file can be opened for sequential input or random access on more than one file number at a time. A file may be opened for output, however, on only one file number at a time.

*Example*   10 OPEN "I",2,"INVEN"

See also the example for the FIELD statement in this chapter.

# OPTION BASE

*Syntax*   OPTION BASE n

*Purpose*   To declare the minimum value for array subscripts.

*Remarks*   n is either 1 or 0. The default value is 0. If the statement

OPTION BASE 1

is executed, the lowest value an array subscript may have is one.

OPTION BASE must be coded before you define or use any arrays.

# POKE

*Syntax*   POKE I,J

*Purpose*   To write a byte into a memory location.

*Remarks*   I and J are integer expressions. The expression I is the address of the memory location and J is the data byte. J must be in the range 0 to 255. I must be in the range 0 to 65536.

The complementary function of POKE is PEEK. The argument to PEEK is an address from which a byte is to be read.

POKE and PEEK are useful for efficiently storing data, loading assembly language subroutines, and passing arguments and results to and from assembly language subroutines.

*Example*   10 POKE 106,0

# PRINT

*Syntax*   PRINT [<list of expressions>]

*Purpose*   To display data at the screen.

*Remarks*   If <list of expressions> is omitted, a blank line is printed. If <list of expressions> is included, the values of the expressions are displayed on the screen. The expressions in the list may be numeric and/or string expressions. (Strings must be enclosed in quotation marks.)

*Print Positions*

The position of each printed item is determined by the punctuation used to separate the items in the list. BASIC divides the line into print zones of 14 spaces each. In the list of expressions, a comma causes the next value to be printed at the beginning of the next zone. A semicolon causes the next value to be printed immediately after the last value. Typing one or more spaces between expressions has the same effect as typing a semicolon.

If a comma or a semicolon terminates the list of expressions, the next PRINT statement begins printing on the same line, spacing accordingly. If the list of expressions terminates without a comma or a semicolon, a <RETURN> is printed at the end of the line. If the printed line is longer than the screen width, BASIC goes to the next physical line to continue printing.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign. Single precision numbers that can be represented with 6 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format are output using the unscaled format. For example, $10^{(-7)}$ is output as .0000001 and $10^{(-8)}$ is output as 1E-08. Double precision numbers that can be represented with 16 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format are output using the unscaled format. For example, 1D-15 is output as .000000000000001 and 1D-16 is output as 1D-16.

A question mark may be used in place of the word PRINT in a PRINT statement.

*Example 1*

```
Ok
10 X=5
20 PRINT X+5, X-5, X*(-5), X ^ 5
30 END
RUN
 10          0          -25          3125
Ok
```

In this example, the commas in the PRINT statement cause each value to be printed at the beginning of the next print zone.

*Example 2*

```
Ok
10 INPUT X
20 PRINT X "SQUARED IS" X^2 "AND";
30 PRINT X "CUBED IS" X^3
40 PRINT
50 GOTO 10
Ok
RUN
? 9
 9 SQUARED IS 81 AND 9 CUBED IS 729

? 21
 21 SQUARED IS 441 AND 21 CUBED IS 9261

?
```

In this example, the semicolon at the end of line 20 causes both PRINT statements to be printed on the same line, and line 40 causes a blank line to be printed before the next prompt.

*Example 3*

```
Ok
10 FOR X = 1 TO 5
20 J=J+5
30 K=K+10
40 ?J;K;
50 NEXT X
Ok
RUN
 5 10 10 20 15 30 20 40 25 50
Ok
```

In this example, the semicolons in the PRINT statement cause each value to be printed immediately after the preceding value. (Don't forget, a number is always followed by a space and a positive number is preceded by a space.) In line 40, a question mark is used instead of the word PRINT.

# *PRINT USING*

**Syntax**    PRINT USING <string exp>;<list of expressions>

**Purpose**    To print strings or numbers using a specified format.

**Remarks**    <string exp> is a string literal (or variable comprised
**and**    of special formatting characters). These formatting
**Examples**    characters (see below) determine the field and the
format of the printed strings or numbers.

<list of expressions> consists of the string
expressions or numeric expressions that are to be
printed, separated by semicolons.

*String Fields*

When PRINT USING is used to print strings, one of
three formatting characters may be used to format
the string field:

"!"    Specifies that only the first character in the
given string is to be printed.

" \n spaces \ "
Specifies that 2+n characters from the
string are to be printed. If the backslashes
are typed with no spaces, two characters
will be printed; with one space, three
characters will be printed, and so on. If
the string is longer than the field, the
extra characters are ignored. If the field
is longer than the string, the string will be
left justified in the field and padded with
spaces on the right.

Example:

```
10 A$="LOOK":B$="OUT"
30 PRINT USING "!";A$;B$
40 PRINT USING " \ \ ";A$;B$
50 PRINT USING " \ \ ";A$;B$;"!!"
RUN
LO
LOOKOUT
LOOK OUT   !!
```

"&"    Specifies a variable length string field. When
the field is specified with "&", the string is
output exactly as input.

Example:

```
10 A$="LOOK":B$="OUT"
20 PRINT USING "!";A$;
30 PRINT USING "&";B$
RUN
LOUT
```

*Numeric Fields*

When PRINT USING is used to print numbers, the
following special characters may be used to format
the numeric field:

#    A number sign is used to represent each
digit position. Digit positions are always
filled. If the number to be printed has fewer
digits than positions specified, the number
will be right justified (preceded by spaces) in
the field.

A decimal point may be inserted at any
position in the field. If the format string
specifies that a digit is to precede the
decimal point, the digit will always be printed
(as 0, if necessary). Numbers are rounded as
necessary.

PRINT USING "##.##";.78
0.78

PRINT USING "###.##";987.654
987.65

PRINT USING "##.##   ";10.2,5.3,66.789,.234
10.20   5.30   66.79   0.23

In the last example, three spaces were inserted at the end of the format string to separate the printed values on the line.

+ A plus sign at the beginning or end of the format string causes the sign of the number (plus or minus) to be printed before or after the number.

— A minus sign at the end of the format field causes negative numbers to be printed with a trailing minus sign.

  PRINT USING  "+##.##      ";-68.95,2.4,55.6,-.9
  -68.95   +2.40   +55.60   -0.90

  PRINT USING "##.##—      ";";-68.95,22.449,
  -7.01   68.95-   22.45   7.01-

** A double asterisk at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks. The ** also specifies positions for two more digits.

  PRINT USING "**#.#      ";12.39,-0.9,765.1
  *12.4   *-0.9   765.1

$$ A double dollar sign causes a dollar sign to be printed to the immediate left of the formatted number. The $$ specifies two more digit positions, one of which is the dollar sign. The exponential format cannot be used with $$. Negative numbers cannot

be used unless the minus sign trails to the right.

  PRINT USING "$$###.##";456.78
   $456.78

**$ The **$ at the beginning of a format string combines the effects of the ** and $$ symbols (see above). Leading spaces are asterisk-filled and a dollar sign is printed before the number. **$ specifies three more digit positions, one of which is the dollar sign.

  PRINT USING "**$##.##";2.34
  ***$2.34

A comma specifies an additional digit position. A comma to the left of the decimal point in a formatting string causes a comma to be printed to the left of every third digit to the left of the decimal point. A comma at the end of the format string is printed as part of the string. The comma has no effect if used with the exponential ($\hat{}\hat{}\hat{}\hat{}$) format.

  PRINT USING "####,.##";1234.5
  1,234.50

  PRINT USING "####.##,";1234.5
  1234.50,

$\hat{}\hat{}\hat{}\hat{}$ Four carets may be placed after the digit position characters to specify exponential format. The four carets allow space for E+xx to be printed. Any decimal point position may be specified. The significant digits are left justified, and the exponent is adjusted. Unless a leading + or trailing + or − sign is specified, one digit position will be used to the left of the decimal point to print a space or a minus sign.

PRINT USING "##.## ^ ^ ^ ^ ";234.56
2.35E+02

PRINT USING ".#### ^ ^ ^ ^ -";888888
.8889E+06

PRINT USING "+.## ^ ^ ^ ^ ";123
+.12E+03

—  An underscore in the format string causes
the next character to be output as a literal
character.

PRINT USING "_!##.##_!";12.34
!12.34!

The literal underscore character itself may
be used by placing two underscore characters
(__) in the format string.

%  If the number to be printed is larger than
the specified numeric field, a percent sign
is printed in front of the number. If rounding
causes the number to exceed the field, a
percent sign is printed in front of the
rounded number.

PRINT USING "##.##";111.22
%111.22

PRINT USING ".##";.999
%1.00

If the number of digits specified exceeds 24,
an "Illegal function call" error results.

# PRINT# and PRINT# USING

**Syntax**  PRINT#<filenumber>,[USING<string exp>;]
<list of exps>

**Purpose**  To write data to a sequential disk file.

**Remarks**  <filenumber> is the number used when the file was
opened for output.

<string exp> is comprised of formatting characters
as described in the previous section (PRINT
USING).

The expressions in <list of expressions> are the
numeric and/or string expressions that will be
written to the file.

PRINT# does not compress data on the disk. An
image of the data is written to the disk just as it
would be displayed on the screen with a PRINT
statement. For this reason, care should be taken to
delimit the data on the disk so that it will be input
correctly from the file.

In the list of expressions, numeric expressions
should be delimited by semicolons. For example:

PRINT#1,A;B;C;X;Y;Z

(If commas are used as delimiters, the extra blanks
that are inserted between print fields will also be
written to the file.)

String expressions must be separated by
semicolons in the list. To format the string
expressions correctly on the file, use explicit
delimiters in the list of expressions.

**Examples**  Let A$="CAMERA" and B$="93604-1". The
statement

PRINT#1,A$;B$

would write CAMERA93604-1 to the file. Because
there are no delimiters, this could not be input as

two separate strings. To correct the problem, insert explicit delimiters into the PRINT# statement as follows:

PRINT#1,A$;",";B$

The image written to the file is

CAMERA,93604-1

which can be read back into two string variables.

If the strings themselves contain commas, semicolons, significant leading blanks, carriage returns, or line feeds, surround them with explicit quotation marks using CHR$(34) before writing them to the file.

For example, let A$="CAMERA, AUTOMATIC" and B$="    93604-1". The statement

PRINT#1,A$;B$

would write the following image to the file:

CAMERA, AUTOMATIC    93604-1

The statement

INPUT#1,A$,B$

inputs "CAMERA" to A$ and "AUTOMATIC 93604-1" to B$. To separate these strings properly in the file, write double quotation marks to the file image using CHR$(34). The statement

PRINT#1,CHR$(34);A$;CHR$(34);CHR$(34);B$;CHR$(34)

writes the following image to the file:

"CAMERA, AUTOMATIC" "    93604-1"

The statement

INPUT#1,A$,B$

inputs "CAMERA, AUTOMATIC" to A$ and " 93604-1" to B$.

The PRINT# statement may also be used with the USING option to control the format of the disk file. For example:

PRINT#1,USING"$$###.##,";J;K;L

For more examples using PRINT#, see the example for the KILL statement and Program 1 in Appendix B.

# PUT

**Syntax**    PUT [#]<file number>[,<record number>]

**Purpose**    To write a record from a random buffer to a random disk file.

**Remarks**    <file number> is the number under which the file was opened and <record number> is the record number for the record to be written.

If the <record number> is omitted, the record will have the next available record number (after the last PUT). The largest possible record number is 32767. The smallest record number is 1.

**Note**

PRINT#, PRINT# USING, and WRITE# may be used to put characters in the random file buffer before a PUT statement.

In the case of WRITE#, BASIC pads the buffer with spaces up to the carriage return. Any attempt to read or write past the end of the buffer causes a "Field overflow" error.

*Example*    See the examples in Appendix B.

# RANDOMIZE

*Syntax*    RANDOMIZE [<expression>]

*Purpose*    To reseed the random number generator.

*Remarks*    <expression> is a numeric expression. If <expression> is omitted, BASIC suspends program execution and asks for a value by printing

Random Number Seed (-32768 to 32767)?

before executing RANDOMIZE.

If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is RUN. To change the sequence of random numbers every time the program is RUN, place a RANDOMIZE statement at the beginning of the program and change the argument with each RUN.

*Example*    Ok
10 RANDOMIZE
20 FOR I=1 TO 5
30 PRINT RND;
40 NEXT I
Ok
RUN
RANDOM NUMBER SEED (-32768 to 32767)?  3

user types 3

.88598  .484668  .586328  .119426  .709225
Ok

RUN
RANDOM NUMBER SEED (-32768 to 32767)?  4

user types 4 for new sequence

.803506  .162462  .929364  .292443  .322921
Ok

RUN
RANDOM NUMBER SEED (-32768 to 32767)?  3

same sequence as first RUN

.88598  .484668  .586328  .119426  .709225
Ok

# READ

*Syntax*    READ <variable list>

*Purpose*    To read values from a DATA statement and assign them to variables.

*Remarks*    A READ statement must always be used in conjunction with a DATA statement. READ statements assign variables to DATA statement values on a one-to-one basis. READ statement variables may be numeric or string, and the values read must agree with the variable types specified. If they do not agree, a "Syntax error" will result.

A single READ statement may access one or more DATA statements (they will be accessed in order), or, several READ statements may access the same DATA statement. If the number of variables in

<variable list> exceeds the number of elements in the DATA statement(s), an "OUT OF DATA" message is printed. If the number of variables specified is fewer than the number of elements in the DATA statement(s), subsequent READ statements will begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

To reread DATA statements from the start, use the RESTORE statement.

*Example 1*

```
.
.
.
80 FOR I=1 TO 10
90 READ A(I)
100 NEXT I
110 DATA 3.08,5.19,3.12,3.98,4.24
120 DATA 5.08,5.55,4.00,3.16,3.37
.
.
.
```

This program segment reads the values from the DATA statements into the array A. After execution, the value of A(1) will be 3.08, and so on.

*Example 2*

```
Ok
10 PRINT "CITY", "STATE", " ZIP"
20 READ C$,S$,Z
30 DATA "DENVER,", COLORADO, 80211
40 PRINT C$,S$,Z
Ok
RUN
CITY            STATE           ZIP
DENVER,         COLORADO        80211
Ok
```

This program reads string and numeric data from the DATA statement in line 30.

# REM

| | |
|---|---|
| *Syntax* | REM <remark> |
| *Purpose* | To allow explanatory remarks to be inserted in a program. |
| *Remarks* | REM statements are not executed but are output exactly as entered when the program is listed. |

REM statements may be branched into or from a GOTO or GOSUB statement; execution continues with the first executable statement after the REM statement.

Remarks may be added to the end of a line by preceding the <remark> with a single quotation mark instead of :REM. <remark> may consist of any sequence of characters.

### Warning

Do not use REM in a data statement, as it is considered legal data.

*Examples*

```
.
.
.
120 REM CALCULATE AVERAGE VELOCITY
130 FOR I=1 TO 20
140 SUM=SUM + V(I)
.
.
.
```

or

.
.
.
```
120 FOR I=1 TO 20 'CALCULATE AVERAGE
VELOCITY
130 SUM=SUM+V(I)
140 NEXT I
```
.
.
.

# *RENUM*

| | |
|---|---|
| *Format* | RENUM [[<new number>][,[<old number>][,<increment>]]] |
| *Purpose* | To renumber program lines. |
| *Remarks* | <new number> is the first line number to be used in the new sequence. The default new number is 10. |

<old number> is the line in the current program where renumbering is to begin. The default old number is the first line of the program.

<increment> is the increment to be used in the new sequence. The default value is 10.

RENUM also changes all line number references following GOTO, GOSUB, THEN, ON...GOTO, ON...GOSUB, and ERL statements to reflect the new line numbers. If a nonexistent line number appears after one of these statements, the error message "Undefined line xxxxx in yyyyy" is printed. The incorrect line number reference xxxxx is not changed by RENUM, but line number yyyyy may be changed.

*Note*

RENUM cannot be used to change the order of program lines (for example, RENUM 15,30 when the program has three lines numbered 10, 20, and 30) or to create line numbers greater than 65529. In such cases, an "Illegal function call" error occurs.

| *Examples* | | |
|---|---|---|
| | RENUM | Renumbers the entire program. The first new line number will be 10. Lines will be numbered in increments of 10. |
| | RENUM 300,,50 | Renumbers the entire program. The first new line number will be 300. Lines will be numbered in increments of 50. |
| | RENUM 1000,900,20 | Renumbers the lines from 900 up so they start with line number 1000 and continue in increments of 20. |

# *RESET*

| | |
|---|---|
| *Syntax* | RESET |
| *Purpose* | To perform a "warm boot" (clears a BDOS R/O error). |
| *Remarks* | Always execute a RESET command after changing diskettes. Otherwise, you will not be able to write to the new diskette. |

RESET also closes all open files. Therefore, when changing diskettes, a CLOSE statement should be executed before removing the old diskette.

# RESTORE

**Syntax**  RESTORE [<line number>]

**Purpose**  To allow DATA statements to be reread beginning at a specified line.

**Remarks**  After RESTORE is executed, the next READ statement accesses the first item in the first DATA statement in the program. If <line number> is specified, the next READ statement accesses the first item in the specified DATA statement.

**Example**
```
Ok
10 READ A,B,C
20 RESTORE
30 READ D,E,F
40 DATA 57, 68, 79
        .
        .
        .
```

# RESUME

**Syntax**
$$\text{RESUME} \left[ \begin{cases} <0> \\ \text{NEXT} \\ <\text{line number}> \end{cases} \right]$$

**Purpose**  To continue program execution after an error recovery procedure has been performed.

**Remarks**  Any one of the four formats shown above may be used, depending upon where execution is to resume:

| | |
|---|---|
| RESUME<br>or<br>RESUME 0 | Execution resumes at the statement that caused the error. |
| RESUME NEXT | Execution resumes at the statement immediately following the one that caused the error. |
| RESUME <line number> | Execution resumes at <line number>. |

A RESUME statement that is not in an error trap routine causes a "RESUME without error" message to be printed.

**Example**
```
Ok
10 ON ERROR GOTO 900
        .
        .
        .
900 IF (ERR=230)AND(ERL=90) THEN PRINT
"TRY AGAIN":RESUME 80
        .
        .
        .
```

# RUN

**Syntax 1**  RUN [<line number>]

**Purpose**  To execute the program currently in memory.

**Remarks**  If <line number> is specified, execution begins on that line. Otherwise, execution begins at the lowest line number. BASIC always returns to command level after a RUN is executed.

**Example**  RUN

**Syntax 2**    RUN <filespec>[,R]

**Purpose**    To load a file from disk into memory and run it.

**Remarks**    <filespec> is a string expression that includes the name used when the file was saved. With CP/M, if no filename extension is given, the default extension .BAS is supplied.

RUN closes all open files and deletes the current contents of memory before loading the designated program. However, with the "R" option, all data files remain OPEN.

**Example**    RUN "NEWFIL",R

See also the programs listed in Appendix B, "Microsoft BASIC Disk I/O."

# *SAVE*

**Syntax**    SAVE <filespec>[, $\begin{Bmatrix} <A> \\ <P> \end{Bmatrix}$ ]

**Purpose**    To save a program file on disk.

**Remarks**    <filespec> is a string expression that includes the name used when the file was saved. With CP/M, if no filename extension is given, the default extension .BAS is supplied. If the <filespec> already exists, the file will be overwritten.

Use the A option to save the file in ASCII format. Otherwise, BASIC saves the file in a compressed binary format. ASCII format takes more space on the disk, but some disk access requires that files be

in ASCII format. For example, the MERGE command requires an ASCII format file.

Use the P option to protect the file by saving it in an encoded binary format. When a protected file is later RUN (or loaded), any attempt to list or edit it will fail.

### *Warning*

Once the P option is used, a file cannot be "unprotected."

**Examples**    SAVE"COM2",A
SAVE"PROG",P

See also the programs listed in Appendix B, "Microsoft BASIC Disk I/O."

# *STOP*

**Syntax**    STOP

**Purpose**    To terminate program execution and return to command level.

**Remarks**    STOP statements may be used anywhere in a program to terminate execution. When a STOP is encountered, the following message is printed:

Break in line nnnnn

Unlike the END statement, the STOP statement does not close files.

BASIC always returns to command level after a STOP is executed. Execution is resumed by issuing a CONT command.

| | |
|---|---|
| *Example* | Ok<br>10 INPUT A,B,C<br>20 K=A^2*5.3:L=B^3/.26<br>30 STOP<br>40 M=C*K+100:PRINT M<br>RUN<br>? 1,2,3<br>BREAK IN 30<br>Ok<br>PRINT L<br> 30.7692<br>Ok<br>CONT<br> 115.9<br>Ok |

## SWAP

| | |
|---|---|
| *Syntax* | SWAP <variable>,<variable> |
| *Purpose* | To exchange the values of two variables. |
| *Remarks* | Any type variable may be swapped (integer, single precision, double precision, string), but the two variables must be of the same type or a "Type mismatch" error results. |
| *Example* | Ok<br>10 A$="ONE" : B$="ALL" : C$="FOR"<br>20 PRINT A$ C$ B$<br>30 SWAP A$, B$<br>40 PRINT A$ C$ B$<br>RUN<br>Ok<br> ONE FOR ALL<br> ALL FOR ONE<br>Ok |

## SYSTEM

| | |
|---|---|
| *Syntax* | SYSTEM |
| *Purpose* | To close all files and return to CP/M command level. |
| *Remarks* | You cannot use Control-C to return to CP/M, as it always returns you to BASIC. |
| *Example* | Ok<br>SYSTEM<br>A> |

## TRON/TROFF

| | |
|---|---|
| *Syntax* | TRON |
| *Syntax* | TROFF |
| *Purpose* | To trace the execution of program statements. |
| *Remarks* | As an aid in debugging, the TRON statement (executed in either the direct or indirect mode) enables a trace flag that prints each line number of the program as it is executed. The numbers appear enclosed in square brackets. The trace flag is disabled with the TROFF statement (or when a NEW command is executed). |
| *Example* | Ok<br>10 K=10<br>20 FOR J=1 TO 2<br>30 L=K + 10<br>40 PRINT J;K;L<br>50 K=K+10 |

```
60 NEXT
70 END
TRON
Ok
RUN
[10] [20] [30] [40] 1 10 20
[50] [60] [30] [40] 2 20 30
[50] [60] [70]
Ok
TROFF
Ok
```

# WHILE...WEND

**Syntax**    WHILE <expression>

.
.
[<loop statements>]

.
.
WEND

**Purpose**    To execute a series of statements in a loop as long as a given condition is true.

**Remarks**    If <expression> is not zero (i.e., true), <loop statements> are executed until the WEND statement is encountered. BASIC then returns to the WHILE statement and checks <expression>. If it is still true, the process is repeated. If it is not true, execution resumes with the statement following the WEND statement.

WHILE/WEND loops may be nested to any level. Each WEND will match the most recent WHILE. An unmatched WHILE statement causes a "WHILE without WEND" error, and an unmatched WEND statement causes a "WEND without WHILE" error.

**Example**
```
90 'BUBBLE SORT ARRAY A$
100 FLIPS=1 'FORCE ONE PASS THRU LOOP
110 WHILE FLIPS
115        FLIPS=0
120        FOR I=1 TO J-1
130             IF A$(I)>A$(I+1) THEN
                     SWAP A$(I),A$(I+1):
                     FLIPS=1
140        NEXT I
150 WEND
```

# WIDTH

**Syntax**    WIDTH [LPRINT]  <integer expression>

**Purpose**    To set the printed line width for the screen or line printer to a specified number of characters.

**Remarks**    The <integer expression> must have a value in the range 15 to 255. The default width is 80 characters.

If the LPRINT option is omitted, the line width is set at the screen. If LPRINT is included, the line width is set at the line printer.

If <integer expression> is 255, the line width is "infinite," that is, BASIC never inserts a carriage return. However, the position of the cursor or the print head, as given by the POS or LPOS function, returns to zero after position 255.

**Example**
```
Ok
10 PRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
RUN
ABCDEFGHIJKLMNOPQRSTUVWXYZ
Ok
```

```
WIDTH 18
Ok
RUN
ABCDEFGHIJKLMNOPQR
STUVWXYZ
Ok
```

# WRITE

| | |
|---|---|
| **Syntax** | WRITE [ <expression>,<expression>, . . . ] |
| **Purpose** | To output data on the screen. |
| **Remarks** | If <expression> is omitted, a blank line is output. If <expression> is included, the values of the expression(s) are output on the screen. The expressions may be numeric and/or string expressions, and they must be separated by commas.

In printed output, each item is separated from the last by a comma. Printed strings are delimited by quotation marks. After the last item in the list is printed, BASIC inserts a carriage return/line feed sequence.

WRITE outputs numeric values using the same format as the PRINT statement. |
| **Example** | Ok<br>10 A=80:B=90:C$="THAT'S ALL"<br>20 WRITE A,B,C$<br>RUN<br>80, 90,"THAT'S ALL"<br>Ok |

# WRITE #

| | |
|---|---|
| **Syntax** | WRITE#<file number>,<expression>, [<expression>,...] |
| **Purpose** | To write data to a sequential file. |
| **Remarks** | <file number> is the number under which the file was opened in "O" mode. The <expressions> may be either string or numeric. They must be separated by commas.

WRITE#, unlike PRINT#, inserts commas between the items as they are written to disk and delimits strings with quotation marks. Therefore, it is not necessary for the user to put explicit delimiters in the list. A carriage return/line feed sequence is inserted after the last item in the list is written to disk. |
| **Example** | Let A$="CAMERA" and B$="93604-1". The statement:

WRITE#1,A$,B$

writes the following image to disk:

"CAMERA","93604-1"

A subsequent INPUT# statement, such as:

INPUT#1,A$,B$

would input "CAMERA" to A$ and "93604-1" to B$. |

**4**

# Microsoft BASIC Functions

# 4

# *Microsoft BASIC Functions*

The intrinsic functions provided by Microsoft BASIC are presented in this chapter. The functions may be called from any program without further definition.

Functions differ from commands and statements in that they cannot be performed by themselves. They must be used in conjunction with either a statement or command. If used with an assignment statement (=), functions must appear on the right side of the = sign. Commands and statements, on the other hand, may be used by themselves (without arguments).

Each function description consists of the following components:

| | |
|---|---|
| Syntax | Shows the correct format for the function. |
| Action | Describes the action the function takes. |
| Remarks | Describes in detail how the function is used; also discusses special conditions when using the function. |
| Example | Shows sample programs or program segments that demonstrate the use of the function. |

Syntax notation for all functions is given in Chapter 1. Numeric and string arguments (where applicable) have been abbreviated as follows:

| | |
|---|---|
| X and Y | Represent any numeric expressions. |
| I and J | Represent integer expressions. |
| X$ and Y$ | Represent string expressions. |

If a floating-point value is supplied where an integer is required, BASIC rounds the fractional portion and uses the resulting integer.

## *ABS*

| | |
|---|---|
| *Syntax* | ABS(X) |
| *Action* | Returns the absolute value of the expression X. |
| *Example* | Ok<br>PRINT ABS(7*(-5))<br> 35<br>Ok |

## *ASC*

| | |
|---|---|
| *Syntax* | ASC(X$) |
| *Action* | Returns a numeric value that is the ASCII code of the first character of the string X$. (See Appendix F for ASCII codes.) |
| *Remarks* | If X$ is null, an "Illegal function call" error is returned.<br><br>See the CHR$ function for ASCII-to-string conversion. |

| | |
|---|---|
| *Example* | Ok |
| | 10 X$ = "TEST" |
| | 20 PRINT ASC(X$) |
| | RUN |
| | 84 |
| | Ok |

# ATN

| | |
|---|---|
| *Syntax* | ATN(X) |
| *Action* | Returns the arctangent of X in radians. |
| *Remarks* | The result is in the range -pi/2 to pi/2. |
| | The calculation of ATN(X) is performed in single precision, regardless of the declared variable type (integer, single precision, or double precision) of X. |
| *Example* | Ok |
| | 10 INPUT X |
| | 20 PRINT ATN(X) |
| | RUN |
| | ? 3 |
| | 1.24905 |
| | Ok |

# CDBL

| | |
|---|---|
| *Syntax* | CDBL(X) |
| *Action* | Converts X to a double precision number. |

| | |
|---|---|
| *Example* | Ok |
| | 10 A = 454.67 |
| | 20 PRINT A;CDBL(A) |
| | RUN |
| | 454.67   454.6700134277344 |
| | Ok |

# CHR$

| | |
|---|---|
| *Syntax* | CHR$(I) |
| *Action* | Returns a string whose one element has ASCII code I. (ASCII codes are listed in Appendix F.) |
| *Remarks* | CHR$ is commonly used to send a special character to the terminal. For instance, the BEL character could be sent (CHR$(7)) as a preface to an error message. |

> **Note**
>
> The Apple III .CONSOLE driver uses specific ASCII codes for special screen functions. Consult your *Apple III Standard Device Drivers Manual* for more information.
>
> See the ASC function for ASCII-to-numeric conversion.

| | |
|---|---|
| *Example* | Ok |
| | PRINT CHR$(66) |
| | B |
| | Ok |

# CINT

| | |
|---|---|
| *Syntax* | CINT(X) |

| | |
|---|---|
| *Action* | Converts X to an integer. |
| *Remarks* | Converts X to an integer by rounding the fractional portion. If X is not in the range -32768 to 32767, an "Overflow" error occurs.<br><br>See the CDBL and CSNG functions for converting numbers to double precision and single precision data types. See also the FIX and INT functions, both of which return integers. |
| *Example* | Ok<br>PRINT CINT(45.67)<br> 46<br>Ok |

# COS

| | |
|---|---|
| *Syntax* | COS(X) |
| *Action* | Returns the cosine of X. |
| *Remarks* | COS is the trigonometric cosine function. X must be in radians. To convert from degrees to radians, multiply by pi/180 (pi = 3.141593).<br><br>The calculation of COS(X) is performed in single precision regardless of the declared variable type (integer, single precision, or double precision) of X. |
| *Example* | Ok<br>10 X = 2*COS(.4)<br>20 PRINT X<br>RUN<br> 1.84212<br>Ok |

# CSNG

| | |
|---|---|
| *Syntax* | CSNG(X) |
| *Action* | Converts X to a single precision number. |
| *Remarks* | See the CINT and CDBL functions for converting numbers to the integer and double precision data types. |
| *Example* | Ok<br>10 A# = 975.3421#<br>20 PRINT A#; CSNG(A#)<br>RUN<br> 975.3421   975.342<br>Ok |

# CVI, CVS, CVD

| | |
|---|---|
| *Syntax* | CVI(<2-byte string>)<br>CVS(<4-byte string>)<br>CVD(<8-byte string>) |
| *Action* | Convert string variable values to numeric variable values. |
| *Remarks* | Numeric values that are read in from a random disk file must be converted from strings back into numbers. CVI converts a 2-byte string to an integer. CVS converts a 4-byte string to a single precision number. CVD converts an 8-byte string to a double precision number.<br><br>See also "MKI\$, MKS\$, MKD\$" in this chapter and "Random Files" in Appendix B. |

*Example*

```
.
.
.
70 FIELD #1,4 AS N$, 12 AS B$, . . .
80 GET #1
90 Y=CVS(N$)
.
.
.
```

# EOF

*Syntax*      EOF(<file number>)

*Action*      Tests for an end-of-file condition.

*Remarks*     <file number> is the number specified in the OPEN statement.

The EOF function returns -1 (true) if the end of a sequential file has been reached. Use EOF to test for an end-of-file condition while inputting, to avoid "Input past end" errors.

With CP/M, the EOF function may be used with random files. If a GET is executed past the end of file, EOF will return a -1. This may be used to find the size of a file using a binary search or other algorithm.

*Example*     
```
Ok
10 OPEN "I",1,"DATA"
20 C=0
30 IF EOF(1) THEN 100
40 INPUT #1,M(C)
50 C=C+1:GOTO 30
.
.
.
```

# EXP

*Syntax*      EXP(X)

*Action*      Calculates the exponential function e.

*Remarks*     EXP returns the mathematical number e raised to the X power. X must be <=87.3365. If EXP overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

The calculation of EXP(X) is performed in single precision, regardless of the declared variable type (integer, single precision, or double precision) of X.

*Example*     
```
Ok
10 X = 5
20 PRINT EXP (X-1)
RUN
 54.5982
Ok
```

# FIX

*Syntax*      FIX(X)

*Action*      Truncates X to an integer.

*Remarks*     FIX(X) is equivalent to the expression SGN(X)*INT(ABS(X)). The difference between FIX and INT is that FIX does not return the next lower number when X is negative, as INT does.

See the INT and CINT functions which also return an integer.

*Examples:*

```
Ok
PRINT FIX(58.75)
 58
Ok

PRINT FIX(-58.75)
-58
Ok
```

# FRE

| | |
|---|---|
| *Syntax* | FRE(0)<br>FRE(X$) |
| *Action* | Returns the number of bytes in memory not being used by BASIC. |
| *Remarks* | Strings in BASIC often have variable lengths. That is, each time you assign a value to a string, its length may change. Strings are also manipulated dynamically. For this reason string space can be scattered or fragmented. |

FRE(" ") forces a reallocation of memory space (otherwise known as "housecleaning," "garbage collection," etc.) before returning the number of free bytes. Housecleaning collects useful data and frees up unused areas of memory once used for strings. The data is compressed so you can use memory space more efficiently.

BASIC initiates housecleaning when all free memory is used up. The process may take 1 to 1 1/2 minutes. Therefore, using FRE(" ") periodically will result in shorter delays for each housecleaning.

Arguments to FRE are dummy arguments.

*Example*

```
Ok
PRINT FRE(0)
 14542
Ok
```

### Note

The actual value returned by the FRE function may be different from the value returned in this example.

# HEX$

| | |
|---|---|
| *Syntax* | HEX$(X) |
| *Action* | Returns a string that represents the hexadecimal value of the decimal argument. |
| *Remarks* | X is rounded to an integer before HEX$(X) is evaluated. |

See the OCT$ function for octal conversion.

*Example*

```
Ok
10 INPUT X
20 A$ = HEX$(X)
30 PRINT X "DECIMAL IS " A$ "
HEXADECIMAL"
RUN
? 32
 32 DECIMAL IS 20 HEXADECIMAL
Ok
```

# INKEY$

| | |
|---|---|
| *Syntax* | INKEY$ |

| | |
|---|---|
| *Action* | Reads a character from the keyboard. |
| *Remarks* | INKEY$ returns either a one-character string containing a character read from the keyboard or a null string if no character is pending at the keyboard. No characters are echoed and all characters are passed through to the program except for Control-C, which terminates the program. |
| *Example* | Ok<br>1000 'TIMED INPUT SUBROUTINE<br>1010 RESPONSE$=""<br>1020 FOR I%=1 TO TIMELIMIT%<br>1030 A$=INKEY$ : IF LEN(A$)=0<br>THEN 1060<br>1040 IF ASC(A$)=13 THEN<br>TIMEOUT%=0 : RETURN<br>1050 RESPONSE$=RESPONSE$+A$<br>1060 NEXT I%<br>1070 TIMEOUT%=1 : RETURN |

# INPUT$

| | |
|---|---|
| *Syntax* | INPUT$(X[ , [#]Y]) |
| *Action* | Returns a string of X characters, read from the keyboard or from file number Y. |
| *Remarks* | X is a string of characters and Y is the file number used in the OPEN statement. File number 0 is used to denote the keyboard.<br><br>If the keyboard is used for input, no characters are echoed and all control characters are passed through except Control-C, which is used to interrupt the execution of the INPUT$ function. |

| | |
|---|---|
| *Example 1* | Ok<br>5 'LIST THE CONTENTS OF A<br>SEQUENTIAL FILE IN HEXADECIMAL<br>10 OPEN"I",1,"DATA"<br>20 IF EOF(1) THEN 50<br>30 PRINT<br>HEX$(ASC(INPUT$(1,#1)));<br>40 GOTO 20<br>50 PRINT<br>60 END |
| *Example 2* | .<br>.<br>.<br>100 PRINT "TYPE P TO PROCEED OR<br>S TO STOP"<br>110 X$=INPUT$(1)<br>120 IF X$="P" THEN 500<br>130 IF X$="S" THEN 700 ELSE 100<br>.<br>.<br>. |

# INSTR

| | |
|---|---|
| *Syntax* | INSTR([I,]X$,Y$) |
| *Action* | Searches for the first occurrence of string Y$ in X$ and returns the position at which the match is found. Optional offset I sets the position for starting the search. |
| *Remarks* | I is a numeric expression in the range 1 to 255. If I>LEN(X$), or if X$ is null, or if Y$ cannot be found, INSTR returns 0. If Y$ is null, INSTR returns I or 1. X$ and Y$ may be string variables, string expressions or string literals.<br><br>If I=0 is specified, the error message "Illegal argument in <line number>" is returned. |

**Example**

```
Ok
10 X$ = "ABCDEB"
20 Y$ = "B"
30 PRINT
INSTR(X$,Y$);INSTR(4,X$,Y$)
RUN
 2  6
Ok
```

# INT

**Syntax**    INT(X)

**Action**    Returns the largest integer <=X.

**Remarks**    See the CINT and FIX functions, which also return integer values.

**Examples**
```
PRINT INT(99.89)
 99
Ok
```

```
PRINT INT(-12.11)
-13
Ok
```

# LEFT$

**Syntax**    LEFT$(X$,I)

**Action**    Returns a string comprised of the leftmost I characters of X$.

**Remarks**    I must be in the range 0 to 255. If I is greater than LEN(X$), the entire string (X$) will be returned. If I=0, the null string (length zero) is returned.

Also see the MID$ and RIGHT$ functions.

**Example**
```
Ok
10 A$ = "BASIC"
20 B$ = LEFT$(A$,5)
30 PRINT B$
RUN
BASIC
Ok
```

# LEN

**Syntax**    LEN(X$)

**Action**    Returns the number of characters in X$.

**Remarks**    Nonprinting characters and blanks are counted.

**Example**
```
Ok
10 X$ = "PORTLAND, OREGON"
20 PRINT LEN(X$)
RUN
 16
Ok
```

# LOC

**Syntax**    LOC(<file number>)

**Action**    Returns the current position in the file.

**Remarks**    With random disk files, LOC returns the record number just read or written from a GET or PUT statement. If the file was opened but no disk I/O has been performed yet, LOC returns a 0.

With sequential files, LOC returns the number of sectors (128 byte blocks) read from or written to the file since it was opened.

*Example*
```
Ok
200 IF LOC(1)>50 THEN STOP
```

# LOF

*Syntax*    LOF(<file number>)

*Action*    Returns the number of records present in the last extent (128 records) read or written. If the file does not exceed one extent, then LOF returns the true length of the file.

*Example*
```
Ok
110 IF NUM%>LOF(1) THEN PRINT
"INVALID ENTRY"
```

# LOG

*Syntax*    LOG(X)

*Action*    Returns the natural logarithm of X.

*Remarks*    X must be greater than zero.

The calculation LOG(X) is performed in single precision, regardless of the declared variable type (integer, single precision, or double precision) of X.

*Example*
```
Ok
PRINT LOG(45/7)
 1.86075
Ok
```

# LPOS

*Syntax*    LPOS(X)

*Action*    Returns the current position of the line printer print head within the line printer buffer.

*Remarks*    LPOS does not necessarily give the physical position of the print head. X is a dummy argument.

*Example*
```
Ok
100 IF LPOS(X)>60 THEN LPRINT CHR$(13)
```

# MID$

*Syntax*    MID$(X$,I [,J])

*Action*    Returns a string of length J characters from X$ beginning with the Ith character.

*Remarks*    I and J must be in the range 1 to 255. If J is omitted, or if there are fewer than J characters to the right of the Ith character, all rightmost characters beginning with the Ith character are returned. If I>LEN(X$), MID$ returns a null string.

If I=0 is specified, the error message "ILLEGAL ARGUMENT IN <line number>" is returned.

Also see the LEFT$ and RIGHT$ functions.

*Example*

```
Ok
10 A$="GOOD"
20 B$="MORNING EVENING AFTERNOON"
30 PRINT A$;MID$(B$,9,7)
Ok
RUN
GOOD EVENING
Ok
```

## MKI$, MKS$, MKD$

*Syntax*    MKI$(<integer expression>)
MKS$(<single precision expression>)
MKD$(<double precision expression>)

*Action*    Converts numeric values to string values.

*Remarks*    Any numeric value that is placed in a random file
buffer with an LSET or RSET statement must be
converted to a string. MKI$ converts an integer to a
2-byte string. MKS$ converts a single precision
number to a 4-byte string. MKD$ converts a double
precision number to an 8-byte string.

See also "CVI, CVS, CVD" in this chapter and
"Random Files" in Appendix B.

*Example*

```
Ok
90 AMT=(K+T)
100 FIELD #1, 8 AS D$, 20 AS N$
110 LSET D$ = MKS$(AMT)
120 LSET N$ = A$
130 PUT #1
   .
   .
   .
```

## OCT$

*Syntax*    OCT$(X)

*Action*    Returns a string which represents the octal value of
the decimal argument.

*Remarks*    X is rounded to an integer before OCT$(X) is
evaluated.

See the HEX$ function for hexadecimal conversion.

*Example*

```
Ok
PRINT OCT$(24)
 30
Ok
```

## PEEK

*Syntax*    PEEK(I)

*Action*    Returns the byte read from the indicated memory
location (I).

*Remarks*    The returned value is an integer in the range 0 to
255. I must be in the range 0 to 65536.

PEEK is the complementary function of the POKE
statement.

*Example*

```
Ok
A=PEEK(&H5A00)
```

## POS

| | |
|---|---|
| *Syntax* | POS(I) |
| *Action* | Returns the current cursor position. |
| *Remarks* | The current horizontal (column) position of the cursor is returned. The returned value is in the range of 1 (the leftmost position) to 80. X is a dummy argument.<br><br>Also see the LPOS function. |
| *Example* | IF POS(X)>60 THEN PRINT CHR$(13) |

## RIGHT$

| | |
|---|---|
| *Syntax* | RIGHT$(X$,I) |
| *Action* | Returns the rightmost I characters of string X$. |
| *Remarks* | If I=LEN(X$), returns X$. If I=0, the null string (length zero) is returned.<br><br>Also see the MID$ and LEFT$ functions. |
| *Example* | Ok<br>10 A$="DISK BASIC"<br>20 PRINT RIGHT$(A$,8)<br>RUN<br>BASIC<br>Ok |

## RND

| | |
|---|---|
| *Syntax* | RND[(X)] |
| *Action* | Returns a random number between 0 and 1. |
| *Remarks* | The same sequence of random numbers is generated each time the program is RUN unless the random number generator is reseeded (see RANDOMIZE). However, X<0 always restarts the same sequence for any given X.<br><br>X>0 or X omitted generates the next random number in the sequence. X=0 repeats the last number generated. |
| *Example* | Ok<br>10 FOR I=1 TO 5<br>20 PRINT INT(RND*100);<br>30 NEXT<br>RUN<br> 24  30  31  51  5<br>Ok |

## SGN

| | |
|---|---|
| *Syntax* | SGN(X) |
| *Action* | Returns the mathematical sign (signum) function. |
| *Remarks* | If X>0, SGN(X) returns 1.<br>If X=0, SGN(X) returns 0.<br>If X<0, SGN(X) returns -1. |
| *Example* | ON SGN(X)+2 GOTO 100,200,300<br><br>branches to 100 if X is negative, 200 if X is 0 and 300 if X is positive. |

# SIN

**Syntax**        SIN(X)

**Action**        Calculates the trigonometric sine function of the angle X.

**Remarks**       Returns the sine of X in radians.

The calculation of SIN(X) is performed in single precision regardless of the declared variable type (integer, single precision, or double precision) of X.

If you want to convert degrees to radians, multiply by pi/180 (pi = 3.141593).

**Example**       Ok
PRINT SIN(1.5)
.997495
Ok

# SPACE$

**Syntax**        SPACE$(X)

**Action**        Returns a string of spaces of length X.

**Remarks**       The expression X is rounded to an integer and must be in the range 0 to 255.

Also see the SPC function.

**Example**       Ok
10 FOR I = 1 TO 5
20 X$ = SPACE$(I)
30 PRINT X$;I
40 NEXT I
RUN
 1
  2
   3
    4
     5
Ok

# SPC

**Syntax**        SPC(I)

**Action**        Prints I spaces on the screen.

**Remarks**       SPC may only be used with PRINT and LPRINT statements. I must be in the range 0 to 255. A ';' is assumed to follow the SPC(I) command.

Also see the SPACE$ function.

**Example**       Ok
PRINT "OVER" SPC(15) "THERE"
OVER                THERE
Ok

# SQR

**Syntax**        SQR(X)

**Action**        Returns the square root of X.

| | |
|---|---|
| *Remarks* | X must be >=0. |

*Example*
```
Ok
10 FOR X = 10 TO 25 STEP 5
20 PRINT X, SQR(X)
30 NEXT
RUN
 10          3.16228
 15          3.87298
 20          4.47214
 25          5
Ok
```

# STR$

*Syntax*   STR$(X)

*Action*   Returns a string representation of the value of X.

*Remarks*   The VAL function is the inverse of the value of X.

*Example*
```
Ok
5 REM ARITHMETIC FOR KIDS
10 INPUT "TYPE A NUMBER";N
20 ON LEN(STR$(N)) GOSUB 30,100,200,300,
400,500
   .
   .
   .
```

# STRING$

*Syntax*   STRING$(I,J)
        STRING$(I,X$)

*Action*   Returns a string of length I whose characters all have ASCII code J or the first character of X$.

---

*Remarks*   I and J are in the range of 0 to 255.

*Example*
```
Ok
10 X$ = STRING$(10,45)
20 PRINT X$ "MONTHLY REPORT" X$
RUN
—————MONTHLY REPORT—————
Ok
```

# TAB

*Syntax*   TAB(I)

*Action*   Tabs to position I on the screen.

*Remarks*   I must be in the range 1 to 255. If the current print position is already beyond space I, TAB goes to that position on the next line. Space 1 is the leftmost position, and the rightmost position is the width minus one.

TAB may only be used in PRINT and LPRINT statements.

*Example*
```
Ok
10 PRINT "NAME" TAB(25) "AMOUNT" : PRINT
20 READ A$,B$
30 PRINT A$ TAB(25) B$
40 DATA "G. T. JONES","$25.00"
RUN
NAME                    AMOUNT

G. T. JONES             $25.00
Ok
```

# TAN

**Syntax**        TAN(X)

**Action**        Calculates the trigonometric tangent of the angle X.

**Remarks**       Returns the tangent of X in radians.

The calculation of TAN(X) is performed in single precision, regardless of the declared variable type (integer, single precision, or double precision) of X.

If the result of a TAN operation overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

**Example**       10 Y = Q*TAN(X)/2

# USR

**Syntax**        USR[<digit>](X)

**Action**        Calls the indicated assembly language subroutine with the argument X.

**Remarks**       <digit> is in the range 0 to 9 and corresponds to the digit supplied with the DEF USR statement for that routine. If <digit> is omitted, USR0 is assumed.

The CALL statement is another way to call an assembly language routine. See Appendix C for more information on using assembly language subroutines.

**Example**       40 B = T*SIN(Y)
                  50 C = USR(B/2)
                  60 D = USR(B/3)
                  .
                  .
                  .

# VAL

**Syntax**        VAL(X$)

**Action**        Returns the numerical value of string X$.

**Remarks**       The VAL function also strips leading blanks, tabs, and linefeeds from the argument string. For example,

VAL(" -3")

returns -3.

See also the STR$ function for numeric-to-string conversion.

**Example**       Ok
                  10 READ NAME$,CITY$,STATE$,ZIP$
                  20 IF VAL(ZIP$)<90000 OR VAL(ZIP$)>96699 THEN PRINT NAME$
                  TAB(25) "OUT OF STATE"
                  30 IF VAL(ZIP$)>=90801 AND VAL(ZIP$)<=90815 THEN
                  PRINT
                  NAME$ TAB(25) "LONG BEACH"
                  .
                  .
                  .

# VARPTR

**Format**

$$VARPTR \begin{Bmatrix} (\text{<variable name>}) \\ (\#\text{<file number>}) \end{Bmatrix}$$

**Action**    Returns the memory address of a variable or file control block.

**Remarks**    For either argument, the returned address is an integer in the range 0 to 65535.

VARPTR(<variable name>) returns the address of the first byte of data identified with a variable. A value must be assigned to <variable name> prior to execution of VARPTR. Otherwise, an "Illegal function call" error results. Any type variable name may be used (numeric, string, array). The address returned is an integer in the range 32767 to -32768. If a negative address is returned, add it to 65536 to obtain the actual address.

VARPTR is usually used to obtain the address of a variable or array so the address may be passed to an assembly language subroutine. A function call of the form VARPTR(A(0)) is usually specified when passing an array, so that the lowest-addressed element of the array is returned.

### Note

All simple variables should be assigned before calling VARPTR for an array, since the addresses of the arrays change whenever a new simple variable is assigned.

VARPTR(#<file number>) is used for sequential files. It returns the starting address of the disk I/O buffer assigned to <file number>. For random files, it returns the address of the FIELD buffer assigned to <file number>.

**Example**    100 X=USR(VARPTR(Y))

# *Appendices*

# Appendix A

# Converting Programs to Microsoft BASIC

If you have programs written in a BASIC other than Microsoft BASIC, some minor adjustments may be necessary before they can be run with Microsoft BASIC. Here are some specific things to look for when converting BASIC programs.

## String Dimensions

Delete all statements that are used to declare the length of strings. A statement such as DIM A$(I,J), which dimensions a string array for J elements of length I, should be converted to the Microsoft BASIC statement DIM A$(J).

Some BASICs require use of a comma or ampersand for string concatenation. Each of these must be changed to a plus sign, which is the operator for Microsoft BASIC string concatenation. In Microsoft BASIC, the MID$, RIGHT$, and LEFT$ functions are used to form substrings out of strings. Forms such as A$(I) to access the Ith character in A$, or A$(I,J) to take a substring of A$ from position I to position J, must be changed as follows:

| Other BASIC | Microsoft BASIC |
| --- | --- |
| X$=A$(I) | X$=MID$(A$,I,1) |
| X$=A$(I,J) | X$=MID$(A$,I,J-I+1) |

If the substring reference occurs on the left side of an

assignment and X$ is used to replace characters in A$, convert as follows:

| Other BASIC | Microsoft BASIC |
| --- | --- |
| A$(I)=X$ | MID$(A$,1,1)=X$ |
| A$(I,J9)=X$ | MID$(A$,I,J-I+1)=X$ |

## Multiple Assignments

Some BASICs allow statements of the form

        10 LET B=C=0

to set B and C equal to zero. Microsoft BASIC would interpret the second equal sign as a logical operator and set B equal to -1 if C equaled 0. Instead, convert this statement to two assignment statements:

        10 C=0:B=0

## Multiple Statements

Some BASICs use a backslash ( \ ) to separate multiple statements on a line. With Microsoft BASIC, be sure all statements on a line are separated by a colon (:).

## MAT Functions

Programs using the MAT functions available in some BASICs must be rewritten using FOR...NEXT loops to execute properly.

# Appendix B

# Microsoft BASIC Disk I/O

Disk I/O procedures for the beginning BASIC user are examined in this appendix. If you are new to BASIC or if you are encountering disk related errors, read through these procedures and program examples to make sure you are using all the disk statements correctly.

Whenever a <filespec> is required in a disk command or statement, refer to "File Naming Conventions" in Chapter 2 to determine how to specify disk files correctly. The CP/M operating system appends a default extension of .BAS to the filename given in a SAVE, RUN, MERGE, or LOAD command.

## Program File Commands

The following is a review of the commands and statements used in program file manipulation.

SAVE <filespec>[,A]    Writes to disk the program that currently resides in memory. Optional A writes the program as a series of ASCII characters. (Otherwise, BASIC uses a compressed binary format.)

LOAD <filespec>[,R]    Loads the program from disk into memory. Optional R runs the program immediately. LOAD always deletes the current contents of memory and closes all files before loading. If R is included, however, open data files are kept open. Thus, programs can be chained or loaded in sections and access the same data files. (LOAD <filespec>, R and RUN <filespec>,R are equivalent.)

RUN <filespec>[,R]    RUN <filespec> loads the program from disk into memory and runs it. RUN deletes the current contents of memory and closes all files before loading the program. If the R option is included, however, all open data files are kept open. (RUN <filespec>,R and LOAD <filespec>,R are equivalent.)

MERGE <filespec>    Loads the program from disk into memory but does not delete the current contents of memory. The program line numbers on disk are merged with the line numbers in memory. If two lines have the same number, only the line from the disk program is saved. After a MERGE command, the "merged" program resides in memory, and BASIC returns to command level.

KILL <filespec>    Deletes the file from the disk. <filespec> may be a program file or a sequential or random access data file.

NAME <old filespec> Changes the name of a disk file.
AS <filename> NAME may be used with program files, random files, or sequential files.

# Protected File

If you wish to save a program in an encoded binary format, use the "Protect" option with the SAVE command.

For example:

    SAVE "MYPROG",P

A program saved this way cannot be listed or edited. You may also want to save an unprotected copy of the program for listing and editing purposes.

# Disk Data Files:
# Sequential and Random I/O

There are two types of disk data files that may be created and accessed by a BASIC program: sequential files and random access files.

## Sequential Files

Sequential files are easier to create than random files but are limited in flexibility and speed when it comes to accessing data. The data that is written to a sequential file is a series of ASCII characters stored, one item after another (sequentially), in the order it is sent and is read back in the same way.

The statements and functions that are used with sequential files are:

| | | |
|---|---|---|
| CLOSE | LOC | PRINT |
| EOF | LOF | PRINT# USING |
| INPUT# | OPEN | WRITE |
| LINE INPUT# | | |

The following program steps are required to create a sequential file and access the data in it:

1. OPEN the file in "O" mode. OPEN "O",#1,"DATA"

2. Write data to the file using the PRINT# statement. (WRITE# may be used instead.) PRINT#1,A$;B$;C$

3. To access the data in the file, you must CLOSE the file and reopen it in "I" mode. CLOSE #1 OPEN "I",#1,"DATA"

4. Use the INPUT# statement to read data from the sequential file into the program. INPUT#1,X$,Y$,Z$

Program 1 is a short program that creates a sequential file, "DATA," from information you input at the keyboard.

*Program 1—Create a Sequential Data File*

```
10   OPEN "O",#1,"DATA"
20   INPUT "NAME";N$
25   IF N$="DONE" THEN END
30   INPUT "DEPARTMENT";D$
40   INPUT "DATE HIRED";H$
50   PRINT#1,N$;",";D$;",";H$
60   PRINT:GOTO 20
RUN
NAME?  MICKEY MOUSE
DEPARTMENT?  AUDIO/VISUAL AIDS
DATE HIRED?  01/12/72

NAME?  SHERLOCK HOLMES
DEPARTMENT?  RESEARCH
DATE HIRED?  12/03/65

NAME?  EBENEEZER SCROOGE
DEPARTMENT?  ACCOUNTING
DATE HIRED?  04/27/81
```

NAME?   SUPER MANN
DEPARTMENT?   MAINTENANCE
DATE HIRED?   08/16/81

NAME? etc.

Now look at Program 2. It accesses the file "DATA" that was created in Program 1 and displays the name of everyone hired in 1981.

*Program 2—Accessing a Sequential File*

```
10   OPEN "1",#1,"DATA"
20   INPUT#1,N$,D$,H$
30   IF RIGHT$(H$,2)="81" THEN PRINT N$
40   GOTO 20
RUN
Ok
EBENEEZER SCROOGE
SUPER MANN
Input past end in 20
Ok
```

Program 2 reads, sequentially, every item in the file. When all the data has been read, line 20 causes an "Input past end" error. To avoid this error, insert line 15 which uses the EOF function to test for the end of file

```
15 IF EOF(1) THEN END
```

and change line 40 to GOTO 15.

A program that creates a sequential file can also write formatted data to the disk with the PRINT# USING statement. For example, the statement

```
PRINT#1,USING"####.##,";A,B,C,D
```

could be used to write numeric data to disk without explicit delimiters. The comma at the end of the format string separates the items in the disk file.

The LOC function, when used with a sequential file, returns the number of sectors that have been written to or read from the file since it was opened. A sector is a 128-byte block of data.

## Adding Data to a Sequential File

If you have a sequential file residing on disk and want to add more data to the end of it, you cannot simply open the file in "O" mode and start writing data. As soon as you open a sequential file in "O" mode, you destroy its current contents.

The following procedure can be used to add data to an existing file called "NAMES."

1.  OPEN "NAMES" in "I" mode.

2.  OPEN a second file called "COPY" in "O" mode.

3.  Read in the data in "NAMES" and write it to "COPY."

4.  CLOSE "NAMES" and KILL it.

5.  Write the new information to "COPY."

6.  Rename "COPY" as "NAMES" and CLOSE.

7.  Now there is a file on disk called "NAMES" that includes all the previous data plus the new data you just added.

Program 3 illustrates this technique. It can be used to create or add onto a file called NAMES. This program also illustrates the use of LINE INPUT# to read strings with embedded commas from the disk file. Remember, LINE INPUT# reads in characters from the disk until it sees a carriage return (it does not stop at quotation marks or commas) or until it has read 255 characters.

*Program 3—Adding Data to a Sequential File*

```
10   ON ERROR GOTO 2000
20   OPEN "I",#1,"NAMES"
30   REM IF FILE EXISTS, WRITE IT TO "COPY"
40   OPEN "O",#2,"COPY"
50   IF EOF(1) THEN 90
60   LINE INPUT#1,A$
70   PRINT#2,A$
80   GOTO 50
90   CLOSE #1
100  KILL "NAMES"
110  REM ADD NEW ENTRIES TO FILE
120  INPUT "NAME";N$
130  IF N$="" THEN 'CARRIAGE RETURN EXITS
INPUT LOOP
140  LINE INPUT "ADDRESS? ";A$
150  LINE INPUT "BIRTHDAY? ";B$
160  PRINT#2,N$
170  PRINT#2,A$
180  PRINT#2,B$
190  PRINT:GOTO 120
200  CLOSE
205  REM CHANGE FILENAME BACK TO "NAMES"
210  NAME "COPY" AS "NAMES"
2000  IF ERR=53 AND ERL=20 THEN OPEN "O",#2,
"COPY":RESUME 120
2010  ON ERROR GOTO 0
```

The error trapping routine in line 2000 traps a "File does not exist" error in line 20. If this happens, the statements that copy the file are skipped, and "COPY" is created as if it were a new file.

## Random Files

Creating and accessing random files requires more program steps than creating and accessing sequential files. However, there are advantages to using random files. One advantage is that random files require less room on the disk, because BASIC stores them in a packed binary format. (A sequential file is stored as a series of ASCII characters.)

The biggest advantage of using random files is that data can be accessed randomly, i.e., anywhere on the disk—it is not necessary to read through all the information from the beginning of the file, as with sequential files. This is possible because the information is stored and accessed in distinct units called records, each of which is numbered.

The statements and functions that are used with random files are:

| Statements | Functions |
|---|---|
| CLOSE | CVD |
| FIELD | CVI |
| GET | CVS |
| LOC | LOF |
| LSET | MKD$ |
| OPEN | MKI$ |
| PUT | MKS$ |
| RSET | |

## Creating a Random File

The following program steps are required to create a random file.

1.  OPEN the file for random access ("R" mode). This example specifies a record length of 32 bytes. If the record length is omitted, the default is 128 bytes.

    Example:

    OPEN "R", 1,"FILE",32

2.  Use the FIELD statement to allocate space in the random buffer for the variables that will be written to the random file.

    Example:

    FIELD #1, 20 AS N$,
    4 AS A$, 8 AS P$

3. Use LSET to move the data into the random buffer. Numeric values must be made into strings when placed in the buffer. To do this, use the "make" functions: MKI$ to make an integer value into a string, MKS$ to make a single precision value into a string, and MKD$ to make a double precision value into a string.

Example:

```
LSET N$=X$
LSET A$=MKS$(AMT)
LSET P$=TEL$
```

4. Write the data from the buffer to the disk using the PUT statement.

Example:

```
PUT #1,CODE%
```

Look at Program 4. It takes information that is input at the terminal and writes it to a random file. Each time the PUT statement is executed, a record is written to the file. The two-digit code that is input in line 30 becomes the record number.

### Note

Do not use a fielded string variable in an INPUT or LET statement. This causes the pointer for that variable to point into string space instead of the random file buffer.

*Program 4 — Create a Random File*

```
10   OPEN "R",#1,"FILE",32
20   FIELD #1,20 AS N$, 4 AS A$, 8 AS P$
30   INPUT "2-DIGIT CODE";CODE%
40   INPUT "NAME";X$
50   INPUT "AMOUNT";AMT
60   INPUT "PHONE";TEL$:PRINT
70   LSET N$=X$
80   LSET A$=MKS$(AMT)
90   LSET P$=TEL$
100   PUT #1,CODE%
110   GOTO 30
```

## Accessing a Random File

The following program steps are required to access a random file:

1. OPEN the file in "R" mode.

Example:

```
OPEN "R", 1,"FILE",32
```

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be read from the file.

Example:

```
FIELD #1 20 AS N$,
4 AS A$, 8 AS P$
```

### Note

In a program that performs both input and output on the same random file, you can often use just one OPEN statement and one FIELD statement.

3.  Use the GET statement to move the desired record into the random buffer.

    Example:

    GET #1,CODE%

4.  The data in the buffer may now be accessed by the program. Numeric values must be converted back to numbers using the "convert" functions: CVI for integers, CVS for single precision values, and CVD for double precision values.

    Example:

    PRINT N$
    PRINT CVS(A$)

Program 5 accesses the random file "FILE" that was created in Program 4. By entering the three-digit code at the keyboard terminal, the information associated with that code is read from the file and displayed.

*Program 5—Access a Random File*

```
10 OPEN "R",#1,"FILE",32
20 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 GET #1, CODE%
50 PRINT N$
60 PRINT USING "$$###.##";CVS(A$)
70 PRINT P$:PRINT
80 GOTO 30
```

The LOC function, when used with random files, returns the "current record number." The current record number is one plus the last record number that was used in a GET or PUT statement. For example, the statement

    IF LOC(1)>50 THEN END

ends program execution if the current record number in file#1 is greater than 50.

Program 6 is an inventory program that illustrates random file access. In this program, the record number is used as the part number. It is assumed the inventory will contain no more than 100 different part numbers. Lines 900-960 initialize the data file by writing CHR$(255) as the first character of each record. This is used later (line 270 and line 500) to determine whether an entry already exists for that part number.

Lines 130-220 display the different inventory functions that the program performs. When you type in the desired function number, line 230 branches to the appropriate subroutine.

*Program 6—Inventory*

```
120 OPEN"R",#1,"INVEN.DAT",39
125 FIELD#1,1 AS F$,30 AS D$, 2 AS Q$,2 AS R$,4 AS P$
130 PRINT:PRINT "FUNCTIONS:":PRINT
135 PRINT 1,"INITIALIZE FILE"
140 PRINT 2,"CREATE A NEW ENTRY"
150 PRINT 3,"DISPLAY INVENTORY FOR ONE PART"
160 PRINT 4,"ADD TO STOCK"
170 PRINT 5,"SUBTRACT FROM STOCK"
180 PRINT 6,"DISPLAY ALL ITEMS BELOW REORDER
LEVEL"
220 PRINT:PRINT:INPUT"FUNCTION";FUNCTION
225 IF (FUNCTION<1)OR(FUNCTION>6) THEN PRINT
"BAD FUNCTION NUMBER":GO TO 130
230 ON FUNCTION GOSUB 900,250,390,480,560,680
240 GOTO 220
250 REM BUILD NEW ENTRY
260 GOSUB 840
270 IF ASC(F$)<>255 THEN INPUT"OVERWRITE";A$:
IF A$<>"Y" THEN RETURN
280 LSET F$=CHR$(0)
290 INPUT "DESCRIPTION";DESC$
300 LSET D$=DESC$
310 INPUT "QUANTITY IN STOCK";Q%
320 LSET Q$=MKI$(Q%)
330 INPUT "REORDER LEVEL";R%
340 LSET R$=MKI$(R%)
350 INPUT "UNIT PRICE";P
```

```
360 LSET P$=MKS$(P)
370 PUT#1,PART%
380 RETURN
390 REM DISPLAY ENTRY
400 GOSUB 840
410 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":
RETURN
420 PRINT USING "PART NUMBER ###";PART%
430 PRINT D$
440 PRINT USING "QUANTITY ON HAND ####";CVI(Q$)
450 PRINT USING "REORDER LEVEL ####";CVI(R$)
460 PRINT USING "UNIT PRICE $$##.##";CVS(P$)
470 RETURN
480 REM ADD TO STOCK
490 GOSUB 840
500 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":
RETURN
510 PRINT D$:INPUT "QUANTITY TO ADD ";A%
520 Q%=CVI(Q$)+A%
530 LSET Q$=MKI$(Q%)
540 PUT#1,PART%
550 RETURN
560 REM REMOVE FROM STOCK
570 GOSUB 840
580 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":
RETURN
590 PRINT D$
600 INPUT "QUANTITY TO SUBTRACT";S%
610 Q%=CVI(Q$)
620 IF (Q%-S%)<0 THEN PRINT "ONLY";Q%;" IN STOCK":
GOTO 600
630 Q%=Q%-S%
640 IF Q%=<CVI(R$) THEN PRINT "QUANTITY NOW";Q%;
"REORDER LEVEL";CVI(R$)
650 LSET Q$=MKI$(Q%)
660 PUT#1,PART%
670 RETURN
680 DISPLAY ITEMS BELOW REORDER LEVEL
690 FOR I=1 TO 100
710 GET#1,I
720 IF CVI(Q$)<CVI(R$) THEN PRINT D$;" QUANTITY";
CVI(Q$) TAB(50) "REORDER LEVEL";CVI(R$)
730 NEXT I
740 RETURN
840 INPUT "PART NUMBER";PART%
850 IF(PART%<1)OR(PART%>100) THEN PRINT
"BAD PART NUMBER":GOTO 840 ELSE GET#1,PART%:
RETURN
890 END
900 REM INITIALIZE FILE
910 INPUT "ARE YOU SURE";B$:IF B$<>"Y"
THEN RETURN
920 LSET F$=CHR$(255)
930 FOR I=1 TO 100
940 PUT#1,I
950 NEXT I
960 RETURN
```

# Appendix C

# BASIC
# Assembly Language Subroutines

All versions of Microsoft BASIC have provisions for interfacing with assembly language subroutines via the USR function and the CALL statement.

The USR function allows assembly language subroutines to be called in the same way BASIC intrinsic functions are called.

## Memory Allocation

Memory space must be set aside for an assembly language subroutine before it can be loaded. During initialization, enter the highest possible memory location minus the amount of memory needed for the assembly language subroutine(s) with the /M: switch.

BASIC uses all memory available from its starting location upwards, so only the topmost locations in memory can be set aside for user subroutines.

If, when an assembly language subroutine is called, more stack space is needed, BASIC's stack can be saved and a new stack set up for use by the assembly language subroutine. BASIC's stack must be restored, however, before returning from the subroutine.

The assembly language subroutine may be loaded into memory by means of the operating system or the BASIC POKE statement.

## USR Function Calls

The format of the USR function is

USR[<digit>](<argument>)

where <digit> is a number from 0 to 9 and the <argument> is any numeric or string expression. <digit> specifies which USR routine is being called and corresponds with the digit supplied in the DEF USR statement for that routine. If <digit> is omitted, USR0 is assumed. The address given in the DEF USR statement determines the starting address of the subroutine.

When the USR function call is made, register A contains a value that specifies the type of argument that was given. The value in A may be one of the following:

| Value in A | Type of Argument |
|---|---|
| 2 | Two-byte integer (two's complement) |
| 3 | String |
| 4 | Single precision floating-point number |
| 8 | Double precision floating-point number |

If the argument is a number, the [H,L] register pair points to the Floating-Point Accumulator (FAC) where the argument is stored.

If the argument is an integer:

FAC-3 contains the lower 8 bits of the argument.

FAC-2 contains the upper 8 bits of the argument.

If the argument is a single precision floating-point number:

FAC-3 contains the lowest 8 bits of mantissa.

FAC-2 contains the middle 8 bits of mantissa.

FAC-1 contains the highest 7 bits of mantissa with leading 1 suppressed (implied). Bit 7 is the sign of the number (0=positive, 1=negative). FAC is the exponent minus 128; the binary point is to the left of the most significant bit of the mantissa.

If the argument is a double precision floating-point number:

FAC-7 through FAC-4 contain four more bytes of mantissa. (FAC-7 contains the lowest 8 bits.)

If the argument is a string, the [D,E] register pair points to 3 bytes called the "string descriptor." Byte 0 of the string descriptor contains the length of the string (0 to 255). Bytes 1 and 2, respectively, are the lower and upper 8 bits of the string starting address in string space.

### Caution

If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy your program this way. To avoid unpredictable results, add + " " to the string literal in the program.

Example:

A$ = "BASIC"+""

This copies the string literal into string space and prevents alteration of program text during a subroutine call.

Usually, the value returned by a USR function is the same type (integer, string, single precision, or double precision) as the argument that was passed to it. However, calling the MAKINT routine returns the integer in [H,L] as the value of the function,

forcing the value returned by the function to be an integer. To execute MAKINT, use the following sequence to return from the subroutine:

```
PUSH    H       ;save value to be returned
LHLD    xxx     ;get address of MAKINT routine
XTHL            ;save return on stack and
                ;get back [H,L]
RET             ;return
```

Also, the argument of the function, regardless of its type, may be forced to an integer by calling the FRCINT routine to get the integer value of the argument in [H,L]. Execute the following routine:

```
        LXI     H       ;get address of subroutine
                        ;continuation
        PUSH    H       ;place on stack
        LHLD    xxx     ;get address of FRCINT
        PCHL
SUB1:......
```

## CALL Statement

User function calls to Z80 assembly language subroutines may be made with the CALL statement (see "CALL," Chapter 3).

### Calling a Z80 Subroutine

A CALL statement with no arguments generates a simple CALL instruction. The corresponding subroutine should return via a simple RET. (CALL and RET are 8080 opcodes — see an 8080 reference manual for details.)

A subroutine CALL with arguments results in a somewhat more complex calling sequence. For each argument in the CALL argument list, a parameter is passed to the subroutine. That parameter is the address of the low byte of the argument. Therefore, parameters always occupy two bytes each, regardless of type.

The method of passing parameters depends upon the number of parameters to pass:

1.  If the number of parameters is less than or equal to three, they are passed in the registers. Parameter 1 will be in HL, 2 in DE (if present), and 3 in BC (if present).

2.  If the number of parameters is greater than 3, they are passed as follows:

    a.  Parameter 1 in HL
    b.  Parameter 2 in DE
    c.  Parameters 3 through n in a contiguous data block. BC will point to the low byte of this data block (i.e., to the low byte of parameter 3).

Note that with this scheme the subroutine must know how many parameters to expect in order to find them. Conversely, the calling program is responsible for passing the correct number of parameters. There are no checks for the correct number or type of parameters.

If the subroutine expects more than three parameters, and there is a need to transfer them to a local data area, use a system subroutine to perform this transfer. The subroutine $AT (listed in the following paragraphs) is called with HL pointing to the local data area, BC pointing to the third parameter, and A containing the number of arguments to transfer (i.e., the total number of arguments minus 2). The subroutine is responsible for saving the first two parameters before calling AT. For example, if a subroutine expects five parameters, it should look like this:

```
SUBR: SHLD  P1      ;SAVE PARAMETER 1
       XCHG
       SHLD  P2      ;SAVE PARAMETER 2
       MVI   A,3     ;NO. OF PARAMETERS LEFT
       LXI   H,P3    ;POINTER TO LOCAL AREA
       CALL  $AT     ;TRANSFER THE OTHER 3
                     ;PARAMETERS
       .
       .
       .
```

[Body of subroutine]

.
.
.

```
       RET           ;RETURN TO CALLER
P1:    DS    2       ;SPACE FOR PARAMETER 1
P2:    DS    2       ;SPACE FOR PARAMETER 2
P3:    DS    6       ;SPACE FOR PARAMETERS 3-5
```

A listing of the argument transfer routine $AT follows.

```
00100  ;              ARGUMENT TRANSFER
00200  ;[B,C]         POINTS TO 3RD PARAM.
00300  ;[H,L]         POINTS TO LOCAL STORAGE FOR
                      ;PARAM 3
00400  ;[A]           CONTAINS THE # OF PARAMS TO
                      ;XFER (TOTAL-2)
00500
00600
00700          ENTRY  $AT
00800  $AT:    XCHG          ;SAVE [H,L] IN [D,E]
00900          MOV    H,B
01000          MOV    L,C    ;[H,L] = PTR TO
                            ;PARAMS
01100  AT1:    MOV    C,M
01200          INX    H
01300          MOV    B,M
01400          INX    H      ;[B,C] = PARAM ADR
01500          XCHG          ;[H,L] POINTS TO
                            ;LOCAL STORAGE
01600          MOV    M,C
01700          INX    H
01800          MOV    M,B
01900          INX    H      ;STORE PARAM IN
                            ;LOCAL AREA
02000          XCHG          ;SINCE GOING BACK
                            ;TO AT1
02100          DCR    A      ;TRANSFERRED ALL
                            ;PARAMS?
02200          JNZ    AT1    ;NO, COPY MORE
02300          RET           ;YES, RETURN
```

When accessing parameters in a subroutine, remember that they are pointers to the actual arguments passed.

### Note

The programmer must match the *number, type,* and *length* of the arguments in the calling program with the parameters expected by the subroutine. This applies to BASIC subroutines, as well as those written in assembly language.

## *Appendix D*

## *Summary of Error Codes and Error Messages*

### Program Errors

| Error Code | Message | Explanation |
|---|---|---|
| 1 | NEXT without FOR | A variable in a NEXT statement does not correspond to any previously executed, unmatched FOR statement variable. |
| 2 | Syntax error | A line is encountered that contains some incorrect sequence of characters (such as unmatched parenthesis, mis-spelled command or statement, incorrect punctuation, etc.). |
| 3 | RETURN without GOSUB | A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement. |
| 4 | Out of data | A READ statement is executed when there are no DATA statements with unread data remaining in the program. |

| 5 | Illegal function call | A parameter that is out of range is passed to a math or string function. This error may also occur as the result of: |
|---|---|---|

1. A negative or unreasonably large subscript

2. A negative or zero argument with LOG

3. A negative argument to SQR

4. A negative mantissa with a non-integer exponent

5. A negative record on a GET or PUT statement

6. A call to a USR function for which the starting address has not yet been given

7. An improper argument to MID$, LEFT$, RIGHT$, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR, or ON...GOTO

| 6 | Overflow | The result of a calculation is too large to be represented in Microsoft BASIC's number format. If underflow occurs, the result is zero and execution continues without an error. |
|---|---|---|
| 7 | Out of memory | A program is too large, has too many FOR loops or GOSUBs, too many variables, or expressions that are too complicated. |
| 8 | Undefined line | A line referenced in a GOTO, GOSUB, IF...THEN...ELSE, or DELETE statement is a nonexistent line. |

| 9 | Subscript out of range | Caused by one of three conditions: |
|---|---|---|

1. An array element is referenced with a subscript that is outside the dimensions of the array

2. An array element is referenced with the wrong number of subscripts.

3. A subscript was used on a variable that is not an array.

| 10 | Redimensioned array | Caused by one of three conditions: |
|---|---|---|

1. Two DIM statements are given for the same array.

2. A DIM statement is given for an array after the default dimension of 10 has been established for that array.

3. An OPTION BASE statement has been encountered after an array has been dimensioned by either default or a DIM statement.

| 11 | Division by zero | Caused by one of two conditions: |
|----|------------------|-----------------------------------|

1. A division by zero operation is encountered in an expression. Machine infinity with the sign of the numerator is supplied as the result of the division. Execution continues.

2. Raising zero to a negative power occurred. Positive machine infinity with the sign of the numerator is supplied as the result of the division. Execution continues.

| 12 | Illegal direct | A statement that is illegal in direct mode is entered as a direct mode command. For example, DEF FN. |
|----|------------------|-----------------------------------|

| 13 | Type mismatch | A string variable name is assigned a numeric value or vice versa; a function that expects a numeric argument is given a string argument or vice versa. This error may also be caused by trying to swap single precision and double precision values. |
|----|------------------|-----------------------------------|

| 14 | Out of string space | String variables have caused BASIC to exceed the amount of free memory remaining. BASIC will allocate string space dynamically, until it runs out of memory. |
|----|------------------|-----------------------------------|

| 15 | String too long | An attempt was made to create a string more than 255 characters long. |
|----|------------------|-----------------------------------|

| 16 | String formula too complex | A string expression is too long or too complex. The expression should be broken into smaller expressions. |
|----|------------------|-----------------------------------|

| 17 | Can't continue | An attempt is made to continue a program that: |
|----|------------------|-----------------------------------|

1. has halted due to an error,

2. has been modified during a break in execution, or

3. does not exist.

| 18 | Undefined user function | A USR function is called before the function definition (DEF statement) is given. |
|----|------------------|-----------------------------------|

| 19 | No RESUME | An error trapping routine is entered, but contains no RESUME statement. |
|----|------------------|-----------------------------------|

| 20 | RESUME without error | A RESUME statement is encountered before an error trapping routine is entered. |
|----|------------------|-----------------------------------|

| 21 | Unprintable error | An error message is not available for the error condition which exists. This is usually caused by an ERROR with an undefined error code. |
|----|------------------|-----------------------------------|

| 22 | Missing operand | An expression contains an operator without a following operand. |
|----|------------------|-----------------------------------|

| 23 | Line buffer overflow | An attempt is made to input a line that has too many characters. |
| 26 | FOR without NEXT | A FOR statement was encountered without a matching NEXT statement. |
| 29 | WHILE without WEND | A WHILE statement was encountered without a matching WEND statment. |
| 30 | WEND without WHILE | A WEND statement was encountered without a matching WHILE statement. |

## Disk Errors

| Error Code | Message | Explanation |
| --- | --- | --- |
| 50 | Field overflow | A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file. |
| 51 | Internal error | An internal malfunction has occurred in Microsoft BASIC. Report to Microsoft the conditions under which the message appeared. |
| 52 | Bad file number | A statement or command references a file with a file number that is not OPEN or is out of the range of file numbers specified at initialization. |
| 53 | File not found | A FILES, LOAD, NAME, or KILL command or OPEN statement references a file that does not exist on the current disk. |
| 54 | Bad file mode | An attempt was made to: 1. use PUT, GET, or LOF with a sequential file 2. LOAD a random file 3. execute an OPEN statement with a file mode other than I, O, or R |
| 55 | File already open | A sequential output mode OPEN is issued for a file that is already open or a KILL is given for a file that is open. |
| 57 | Disk I/O error | An I/O error occurred on a disk I/O operation. It is a fatal error; i.e., the operating system cannot recover from the error. |
| 58 | File already exists | The filename specified in a NAME statement is identical to a filename already in use on the disk. |
| 61 | Disk full | All disk storage space is in use. |
| 62 | Input past end | An INPUT statement is executed after all the data in the file has been INPUT, or for a null (empty) file. To avoid this error, use the EOF function to detect the end of file. |
| 63 | Bad record number | In a PUT or GET statement, the record number is either greater than the maximum allowed (32767) or equal to zero. |
| 64 | Bad file name | An illegal form is used for the filespec with a LOAD, SAVE, or KILL command or an OPEN statement (e.g., a filename with too many characters). |

| 66 | Direct statement in file | A direct statement is encountered while loading an ASCII-format file. The LOAD operation is terminated. |
| 67 | Too many files | An attempt is made to create a new file (using SAVE or OPEN) when all 255 directory entries are full. |

# Appendix E

# Mathematical Functions

## Derived Functions

Functions that are not intrinsic to Microsoft BASIC may be calculated as follows.

| Function | Microsoft BASIC Equivalent |
|---|---|
| SECANT | SEC(X)=1/COS(X) |
| COSECANT | CSC(X)=1/SIN(X) |
| COTANGENT | COT(X)=1/TAN(X) |
| INVERSE SINE | ARCSIN(X)=ATN(X/SQR(-X*X+1)) |
| INVERSE COSINE | ARCCOS(X)=-ATN (X/SQR(-X*X+1)) +1.5708 |
| INVERSE SECANT | ARCSEC(X)=ATN(X/SQR(X*X-1)) +SGN(SGN(X)-1)*1.5708 |
| INVERSE COSECANT | ARCCSC(X)=ATN(X/SQR(X*X-1)) +(SGN(X)-1)*1.5708 |
| INVERSE COTANGENT | ARCCOT(X)=ATN(X)+1.5708 |
| HYPERBOLIC SINE | SINH(X)=(EXP(X)-EXP(-X))/2 |
| HYPERBOLIC COSINE | COSH(X)=(EXP(X)+EXP(-X))/2 |
| HYPERBOLIC TANGENT | TANH(X)=EXP(-X)/EXP(X)+ EXP(-X)*2+1 |
| HYPERBOLIC SECANT | SECH(X)=2/(EXP(X)+EXP(-X)) |

| | |
|---|---|
| HYPERBOLIC COSECANT | CSCH(X)=2/(EXP(X)-EXP(-X)) |
| HYPERBOLIC COTANGENT | COTH(X)=EXP(-X)/(EXP(X) -EXP(-X))*2+1 |
| INVERSE HYPERBOLIC SINE | ARCSINH(X)=LOG(X+SQR(X*X+1)) |
| INVERSE HYPERBOLIC COSINE | ARCCOSH(X)=LOG(X+SQR(X*X-1)) |
| INVERSE HYPERBOLIC TANGENT | ARCTANH(X)=LOG((1+X)/(1-X))/2 |
| INVERSE HYPERBOLIC SECANT | ARCSECH(X)=LOG((SQR(-X*X+1) +1)/X) |
| INVERSE HYPERBOLIC COSECANT | ARCCSCH(X)=LOG((SGN(X)*SQR (X*X+1)+1)/X) |
| INVERSE HYPERBOLIC COTANGENT | ARCCOTH(X)=LOG((X+1)/(X-1))/2 |

# *Appendix F*

# *ASCII Character Codes*

| Dec | Hex | CHR | Dec | Hex | CHR | Dec | Hex | CHR |
|---|---|---|---|---|---|---|---|---|
| 000 | 00H | NUL | 028 | 1CH | FS | 056 | 38H | 8 |
| 001 | 01H | SOH | 029 | 1DH | GS | 057 | 39H | 9 |
| 002 | 02H | STX | 030 | 1EH | RS | 058 | 3AH | : |
| 003 | 03H | ETX | 031 | 1FH | US | 059 | 3BH | ; |
| 004 | 04H | EOT | 032 | 20H | SPACE | 060 | 3CH | < |
| 005 | 05H | ENQ | 033 | 21H | ! | 061 | 3DH | = |
| 006 | 06H | ACK | 034 | 22H | " | 062 | 3EH | > |
| 007 | 07H | BEL | 035 | 23H | # | 063 | 3FH | ? |
| 008 | 08H | BS | 036 | 24H | $ | 064 | 40H | @ |
| 009 | 09H | HT | 037 | 25H | % | 065 | 41H | A |
| 010 | 0AH | LF | 038 | 26H | & | 066 | 42H | B |
| 011 | 0BH | VT | 039 | 27H | ' | 067 | 43H | C |
| 012 | 0CH | FF | 040 | 28H | ( | 068 | 44H | D |
| 013 | 0DH | CR | 041 | 29H | ) | 069 | 45H | E |
| 014 | 0EH | SO | 042 | 2AH | * | 070 | 46H | F |
| 015 | 0FH | SI | 043 | 2BH | + | 071 | 47H | G |
| 016 | 10H | DLE | 044 | 2CH | , | 072 | 48H | H |
| 017 | 11H | DC1 | 045 | 2DH | — | 073 | 49H | I |
| 018 | 12H | DC2 | 046 | 2EH | . | 074 | 4AH | J |
| 019 | 13H | DC3 | 047 | 2FH | / | 075 | 4BH | K |
| 020 | 14H | DC4 | 048 | 30H | 0 | 076 | 4CH | L |
| 021 | 15H | NAK | 049 | 31H | 1 | 077 | 4DH | M |
| 022 | 16H | SYN | 050 | 32H | 2 | 078 | 4EH | N |
| 023 | 17H | ETB | 051 | 33H | 3 | 079 | 4FH | O |
| 024 | 18H | CAN | 052 | 34H | 4 | 080 | 50H | P |
| 025 | 19H | EM | 053 | 35H | 5 | 081 | 51H | Q |
| 026 | 1AH | SUB | 054 | 36H | 6 | 082 | 52H | R |
| 027 | 1BH | ESCAPE | 055 | 37H | 7 | 083 | 53H | S |

| 084 | 54H | T | 099 | 63H | c | 114 | 72H | r |
|-----|-----|---|-----|-----|---|-----|-----|---|
| 085 | 55H | U | 100 | 64H | d | 115 | 73H | s |
| 086 | 56H | V | 101 | 65H | e | 116 | 74H | t |
| 087 | 57H | W | 102 | 66H | f | 117 | 75H | u |
| 088 | 58H | X | 103 | 67H | g | 118 | 76H | v |
| 089 | 59H | Y | 104 | 68H | h | 119 | 77H | w |
| 090 | 5AH | Z | 105 | 69H | i | 120 | 78H | x |
| 091 | 5BH | [ | 106 | 6AH | j | 121 | 79H | y |
| 092 | 5CH | \ | 107 | 6BH | k | 122 | 7AH | z |
| 093 | 5DH | ] | 108 | 6CH | l | 123 | 7BH | { |
| 094 | 5EH | ^ | 109 | 6DH | m | 124 | 7CH | \| |
| 095 | 5FH | — | 110 | 6EH | n | 125 | 7DH | } |
| 096 | 60H | ' | 111 | 6FH | o | 126 | 7EH | ~ |
| 097 | 61H | a | 112 | 70H | p | 127 | 7FH | DEL |
| 098 | 62H | b | 113 | 71H | q |  |  |  |

Dec=decimal          Hex=hexadecimal (H)          CHR=character
LF=Line Feed          FF=Form Feed                  CR=Carriage Return
DEL=Rubout

### Note

All 128 ASCII codes can be generated on the Apple III keyboard. However, the .CONSOLE device driver which controls the keyboard allows you to redefine the keys. Your Apple III may be initially set up to use some of the ASCII codes in a different manner than is listed here. Consult your Apple III *Standard Device Drivers Manual* to verify ASCII codes used for the Apple III keyboard.

## Appendix G

## Microsoft BASIC Reserved Words

The following is a list of reserved words used in Microsoft BASIC.

| | | | |
|-----|-------|--------|----------|
| ABS | DIM | INT | OPTION |
| AND | EDIT | KILL | OR |
| ASC | ELSE | LEFT$ | PEEK |
| ATN | END | LEN | POKE |
| AUTO | EOF | LET | POS |
| CALL | ERASE | LINE | PPRINT |
| CDBL | ERL | LIST | PRINT# |
| CHAIN | ERR | LLIST | PUT |
| CHR$ | ERROR | LOAD | RANDOMIZE |
| CINT | END | LOC | READ |
| CLEAR | EXP | LOF | REM |
| CLOSE | FIELD | LOG | RENUM |
| COMMON | FILES | LPOS | RESET |
| CONT | FIX | LPRINT | RESTORE |
| COS | FOR | LSET | RESUME |
| CSNG | FRE | MERGE | RIGHT$ |
| CVD | GET | MID$ | RND |
| CVI | GOSUB | MKD$ | RSET |
| CVS | HEX$ | MKI$ | RUN |
| DATA | IF | MKS$ | SAVE |
| DEFDBL | IMP | MOD | SBN |
| DEF FN | INKEY$ | NAME | SIN |
| DEFINT | INP | NEW | SPACE |
| DEFSNG | INPUT | NOT | SPC |
| DEFSTR | INPUT# | OCT$ | SQR |
| DEF USR | INPUT$ | ON | STOP |
| DELETE | INSTR | OPENON | STR$ |

| STRING$ | THEN | USR | WHILE |
|---------|------|-----|-------|
| SWAP | TO | VAL | WRITE |
| SYSTEM | TROFF | VARPTR | WRITE# |
| TAB | TRON | WAIT | XOR |
| TAN | USING | WEND | |

**Note**

INP, OUT, and WAIT are reserved words, but are not implemented in the SoftCard III version of Microsoft BASIC. Use of these words can result in an error message.

# Index